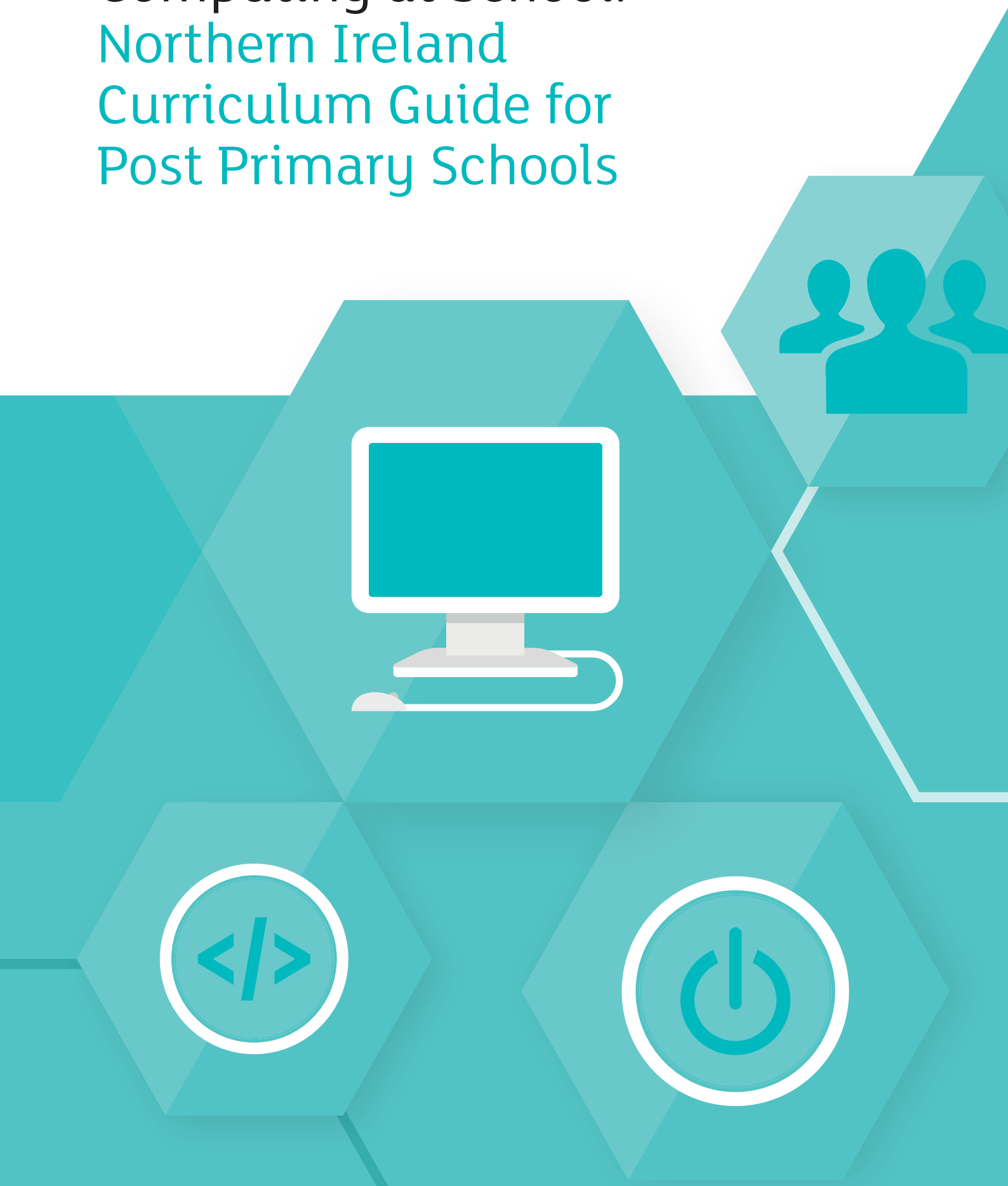
Rewarding Learning

Computing at School: Northern Ireland Curriculum Guide for Post Primary Schools



Contents

Part One

Introduction

Part Two
Introduction

Preface

This guide clarifies how teachers in post-primary schools in Northern Ireland can provide opportunities for their pupils to develop skills in computing and coding.

Part One introduces key concepts in computational thinking and computing. Also published as a separate guide for use in primary schools, it provides the foundations for the more advanced approaches that follow in Part Two. Unless post-primary teachers are already proficient in computational thinking and computing, we recommend that they read Part One first.

Part One – Primary

Content adapted from:
Berry, M., QuickStart Computing: a CPD toolkit for primary teachers, Swindon: BCS.

A catalogue record for this title is available from the British Library.
ISBN: 978-1-78339-521-7

http://primary.quickstartcomputing.org/resources/pdf/qs_handbook.pdf

Introduction

The content in this guide has been adapted from *QuickStart Computing: a CPD toolkit* for primary pupils[1], a resource created by Computing at School (part of BCS, the Chartered Institute for IT).

Computers play a vital role in our lives. From home to work and from research to leisure, there's scarcely a part of modern life untouched by computer technology. Yet, rather than simply become proficient users of computers, it's vital that our pupils learn how such technology works.

Programming, including learning to code, is enjoyable and empowering to learn, as well as rewarding to teach. Unlike other subjects, however, it is one that few teachers studied at school or during their teacher training.

This guide is provided for teachers as an introduction to computational thinking and programming/coding. We hope that the information and support here help you identify and plan Interactive Design activities that support the Explore element of Using ICT, that is, to 'investigate, make predictions and solve problems through interaction with digital tools'.

To further support those schools interested in developing computing and coding activities for their pupils, CCEA will publish a new primary teaching and learning unit specifically written to match the requirements of the Northern Ireland curriculum and the need to develop pupils' digital skills in this area.

This additional resource will outline a variety of age-appropriate activities for each year group in primary school. It will also guide ICT co-ordinators in how to plan for whole-school progression in coding and clarify how schools can embed this significant 21st century skill in their curriculum.

Programming and coding in the Northern Ireland curriculum

CCEA is currently developing a new Digital Skills Framework which will incorporate the statutory 5'E's of Using ICT for primary and post-primary pupils and outline digital progression from Foundation Stage to A level and beyond. It is likely that this framework will include a strand explicitly describing a progression of programming and computer science skills for primary schools.

Computational thinking as defined in this guide overlaps extensively with the Northern Ireland Curriculum requirements to develop pupils' Thinking Skills and Personal Capabilities (TSPC).

[1] *Berry, M (2015) QuickStart Primary Handbook. Swindon: BCS.*

Section 1
Computational thinking

Understanding problems for computers to provide solutions

Computers are incredible: they greatly expand our mental capacities. We can work faster, process more information and share ideas with people across the globe.

There are two steps to solving a problem with a computer:

1. Think about the necessary steps to solve the problem.
2. Use your technical skills to get the computer to work on the problem.

Think, for example, about using a calculator to solve a word problem in Maths. You need to understand the problem before your calculator can help out with the arithmetic.

Similarly, when making an animation, you have to plan your story and how you'll shoot it before your computer hardware and software can help you do the work.

Note: definitions of the main computing terms throughout this guide are provided in the glossary on page 61.

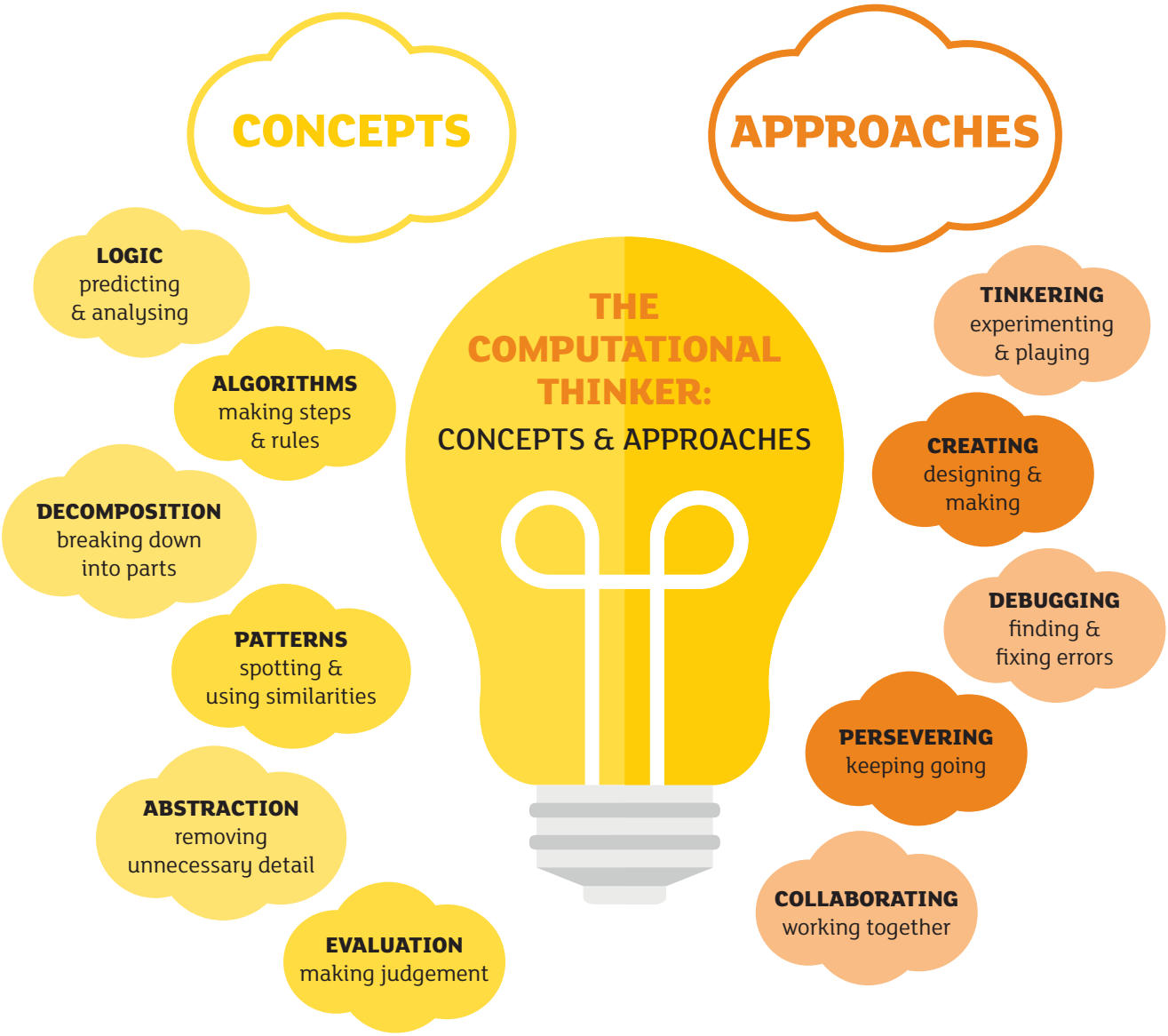
In both examples, the thinking done before starting work on a computer is called computational thinking.

Computational thinking is the term for the processes and approaches used when thinking about problems or systems. It's about considering a problem in ways that mean a computer can help to solve it.

This term was popularised by Jeanette Wing, an American computer scientist. In 2006, she argued that computational thinking should be part of every child's education along with reading, writing and numeracy.

The following are processes used in computational thinking:

- Logical reasoning: predicting and analysing (see pages 11–13)
- Algorithms: making steps and rules (see pages 14–18)
- Decomposition: breaking things down into parts (see pages 19–22)
- Abstraction: removing unnecessary detail (see page 23)
- Patterns and generalisation: noticing and using similarities (see pages 24–26)
- Evaluation: forming judgements



How is computational thinking used?

While computational thinking describes modes of thought that are typical of computer scientists and software developers, many others think this way too – and not just when using computers. The thinking processes and approaches that help with computing are helpful in many other contexts.

For example, the ways in which software engineers go about creating a new social networking platform are not really so different to how you and your colleagues might work as a team to put on a play or organise a trip.

In each case, you must:

- work out the rules or the steps to take for getting things done;
- reflect on the ways in which previous projects were done might help;
- divide the big and complex problem into smaller, more manageable, problems; and
- manage the task's complexity, typically, by focusing on the key details.

How is computational thinking used in a school curriculum?

Logical reasoning, algorithms, decomposition, abstraction and generalisation – all these aspects of computational thinking can help with problem solving across the school curriculum and beyond. As pupils learn these thinking skills in the context of computing work, they usually grow more proficient in applying them to other areas of study.
Your pupils already use computational thinking in many ways and across different curriculum subjects.

Consider these examples.

- When pupils write stories for Language and Literacy, you encourage them to plan first, think about the main narrative events, the settings and the characters.
- In the Arts, you might ask pupils to think about what they are going to create, what steps are needed to create this and how they will work through them. The complex creative process is likely to be broken down into more manageable planned phases.
- When working on a problem in Maths, pupils identify the key information first, before they solve it.

Where does computational thinking fit in the NI Curriculum?

The Northern Ireland Curriculum has prioritised thinking and problem-solving since its introduction in 2007.

At the heart of the curriculum lies an explicit emphasis on the development of skills and capabilities for lifelong learning and for contributing effectively to society. These whole curriculum skills and capabilities consist of the Cross-Curricular Skills (including Using ICT) and Thinking Skills and Personal Capabilities.

Computational thinking, however, shouldn't be seen as an alternative for 'problem-solving skills'. While it does help to solve problems and it does have wide applications across other disciplines, computational thinking is most evident and probably most effectively learned, through the systematic, creative processes of writing code. This is discussed in the second section of this guide.

While programming (see page 31) is an important part of computing, it would be wrong to see this as an end in itself. Rather, it's through the practical experience of programming that the insights of computational thinking can best be developed.

Classroom activities to develop computational thinking

1

Computational sandwiches

Pupils make a recipe for a sandwich; they consider each step in the process carefully. Teach them that a step-by-step sequence of instructions is called an algorithm. Invite them to share recipes and identify patterns in them – this is called generalisation. Read a range of recipes. Discuss the layers of simplification (abstraction) even in simple recipes, such as those for pizza.

2

Extension work

Challenge able or older pupils, either working to individually or collaboratively, to carry out more complex projects. Examples include researching and writing up aspects of a curriculum topic such as the Viking invasion covered in History, or putting together a class play or a school assembly. In each case, ask pupils to note each step needed for the task and to identify any elements they had to leave out to make the subject matter meet the brief.

Logical reasoning

Explain why something happens

Set up two computers in the same way, give them the same instructions (the program) and the same input, and you can be sure they'll produce the same output.

Computers don't invent things on their own initiative, or work differently because of personal feelings. This means that they are predictable. Because of this we can use logical reasoning to predict exactly what a program or computer system will do.

Children learn this quickly. Watching others use computers and experimenting for themselves allow even the very young to understand how technology works. A child soon recognizes that clicking a button brings up a list of different games to play, or that tapping or stroking the screen produce predictable responses.

Using existing knowledge of a system to predict future behaviour is an important part of logical reasoning. Central to logical reasoning is the ability to explain why something is the way it is. And it's a way to understand why something isn't the way you expected it to be.

Using logical reasoning

Logic is crucial to the operation of any computer. Deep inside its central processing unit (CPU), every operation the computer performs is reduced to logical operations carried out using electrical signals.

Because everything a computer does is controlled by logic, we can use logic to understand program behaviour.

Logical reasoning is used continuously by software engineers. They use their specialist understanding of how computer hardware, the relevant operating system (such as Windows 8, OS X) and the programming language they're using work so that they can develop new code that will work in the way they want. They also depend on logical reasoning when they test new software and when they seek out and fix the 'bugs' (mistakes) in their thinking (see page 52) or their coding when such tests fail.

Logical reasoning across the curriculum

Children already use logical reasoning across the wider curriculum in many ways.

- In Language and Literacy, pupils predict what a character will do next in a novel, or explain the character's actions in the story so far.
- In Science and Technology, pupils explain how conclusions have been drawn from the results of their experiments.
- In History, pupils discuss the logical connections between cause and effect; they understand that historical knowledge is derived from various sources.

Where does logical reasoning fit into computing activities?

Younger pupils can use logical reasoning to predict the behaviour of simple programs. This can include programs they write themselves, for example those made with a floor turtle, or simple movement commands on screen in a program like Scratch. Other uses of logical reasoning include predicting what happens when playing a computer game, or when using a painting program.

Older pupils can write simple algorithms and code using logical reasoning. They can identify and fix mistakes in their own code or in existing code.

Classroom activities to illustrate logical reasoning

1

Floor turtles

The pupils predict where the turtle device will end up when the go button is pressed. Then they explain why they have made this prediction. Being able to explain thinking is what logical reasoning is all about.

2

Debugging

Logical reasoning is crucial in the process of debugging (identifying and correcting errors in coding). Pupils could test programs made on Scratch or Scratch Jnr to learn this. (See https://scratch.mit.edu/ or http://www.scratchjr.org/) The pupils look at one another's Scratch programs and search for bugs. Encourage them to test the programs to see if they can work out exactly which part of the code is causing a problem. If pupils' programs don't work, ask them to talk through their code, explaining it to a classmate. The action of explaining the process is the important part, so it could even be explained to an inanimate object, such as a rubber duck. This is known as rubber duck debugging.

3

Cracking the Code

Print out some of the rules and regulations that apply to your school, for example its Computer Acceptable Use Policy. Ask pupils to think carefully about some specific rules. By using logical reasoning, the pupils have to explain why the rules are as they are.

4

Reasons for Rules

Give pupils a program that you have made, or one you have downloaded from the Scratch website. The pupils must work backwards from the code and work out what it will do.

5

Games and Logic

Many games draw on the players' skills in making logical predictions. Why not use some to help build their skills? First set up noughts and crosses for the pupils to play using pencil and paper. During the game, each pupil should predict their opponent's next move. After this, set up some computer games – Minesweeper, Angry Birds and SimCity all work well. At certain points during gaming, pupils must pause and predict what will happen when they make their next move. A great game to allow pupils to practise their skills of logical reasoning is chess. You might want to start a chess club if there isn't one already in your school.

Google’s search algorithm is said to be a more closely guarded secret than the recipe for Coca-Cola.

Algorithms

A sequence of instructions, or a set of rules for achieving a result or solving a problem, is known as an algorithm.

For example, you probably know the easiest way to get home from school; this might be turn left, drive for five miles, turn right. This might be considered an algorithm – it's a sequence of instructions for achieving a result: to get you to your chosen destination easily. There are plenty of algorithms that will accomplish the same goal (other routes); in this case, there are even algorithms (such as in those programmed into your satnav) for determining the shortest or quickest route.

Algorithms in the real world

Search engines use algorithms to organise search results, with the aim of putting the result you're looking at the top of the page. Google's search algorithm is said to be a more closely guarded secret than the recipe for Coca-Cola.

Online retailers use different types of algorithms. One type works on the basis of other people's purchases. Once you buy an item, the retailer suggests other purchases based on what other people, who bought the same item as you, went on to buy. A different algorithm can be used to drop the price of a new or untried product every few days or every few hours until the product is purchased by someone; after this, the price goes up.

Credit approvals, store cards, job and dating matches and more are all run on similar principles. The most complicated algorithms are found in science, where they are used to design new drugs or model the climate.

Algorithms across the curriculum

Developing pupils' understanding of algorithms could extend throughout the curriculum. Consider the following examples of ways in which you and your pupils might be using them already.

- A recipe can be considered an algorithm.
- A lesson plan is an algorithm for the sequence of events, activities and intended outcomes in a lesson.
- For many activities there is a sequence of steps or rules for pupils to follow, for example when going for lunch or preparing for a PE lesson.
- In Language and Literacy, we might consider the rules for producing writing in a particular form and then for proofreading and redrafting as a type of algorithm.
- In Science, the method of an experiment could be considered an algorithm.
- Your approach to teaching mental arithmetic might be to implement a simple algorithm.

An example of such an algorithm is the following.

- repeat ten times:
  - ask a question;
  - wait for a response; and
  - provide feedback on whether the response was right or wrong.

Where do algorithms fit in with the 5'E's?

The Northern Ireland curriculum requirements for Using ICT are set out under headings often described as the 5'E's. One of the headings, Explore, includes the requirement for pupils to 'investigate, make predictions and solve problems through interaction with digital tools'.

There can be many algorithms to solve the same problem, and each of these can be implemented using different programming languages and on different computer systems. For example, Key Stage 1 pupils might usefully compare how they draw a square with a floor turtle and how they would do this on screen in Logo or ScratchJr.

Scratch Jnr

Key Stage 2 builds on this, and, for CCEA Using ICT tasks, pupils can design Scratch and Logo programs with particular goals in mind. Such work draws on their ability to think algorithmically. Pupils also use logical reasoning to explain algorithms and to detect and correct errors in them.

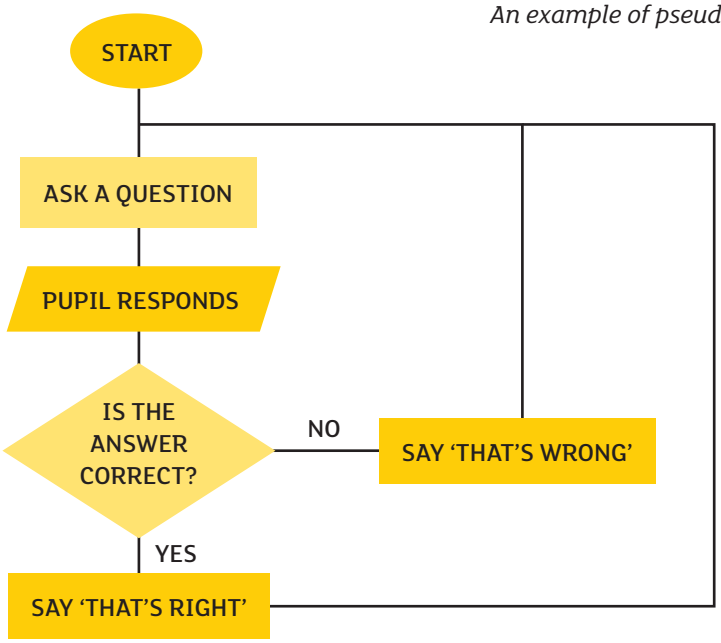
To practise this, encourage pupils to carry out the steps for an algorithm. They must follow the instructions themselves rather than write these as code for a computer. You are likely to see errors and inconsistencies early in the process!

While programming languages like Scratch can minimise the need for a thorough planning stage when writing a program, it's good practice for pupils to prepare thoroughly. They should write down the algorithm for their program. **This doesn't have to be lengthy writing: they can use rough jottings, a storyboard, pseudocode or a flow chart**. (Pseudocode, as seen in the first illustration below, is a written description of how a program will operate; the second illustration, below, is an example of a flow chart.) This type of writing – in whatever form you choose – makes it easier for pupils to receive feedback on their algorithms before implementing these as code on their computers. Feedback can come from other pupils as well as from you.

Repeat 10 times:
Ask a maths question

If the answer is right then:
Say *'well done!'*
Else:
Say *'think again!'*

An example of pseudocode



An example of a flow chart

Classroom activities to illustrate algorithms

1

Discussion

Discuss what makes one algorithm better than another with your pupils. In early programming lessons, pupils should realise that a Bee-Bot program with fewer steps than another to get to the same place is faster.

2

Guess my number

Play 'Guess my number' to demonstrate the point. Tell the pupils that you have picked a number between 1 and 100 which they have to guess. They can ask questions about the number, but you can only answer 'Yes' or 'No'. Each pupil may ask only one question.

- For the first turn, ask the pupils to guess numbers at random.
- Next, using a new number, ask the pupils to guess the number sequentially, beginning with number one. For example, 'Is the number one?' and so on. Explain that this is called a linear search. Give the class as many turns as they need to guess the number.
- Finally, using a new number, explain the term binary search. Explain that the pupils already know the number is less than 100, so suggest that they ask, 'Is it less than 50?' then, 'Is it less than 25?' or 'Is it less than 75?' depending on the answer. The pupils should keep halving the section they are searching in until they find the number.
- After the game, discuss which approach found the number quickest. When they are familiar with the binary search method, replay the game using a number between 1 and 1000.

3

Sorting weights

The pupils sort a set of unknown weights in order of weight using a simple pan balance. They think carefully about the algorithm they're following to do this. Then they think of a quicker way to accomplish the same task. A demonstration of this may be found at http://csunplugged.org/sorting-algorithms

4

Follow the rules

Explain that not all algorithms are made up of sequences of instructions. Some are based on rules. Write a number sequence on the board to introduce this idea, for example 3, 6, 9, 12 or 2, 4, 8, 16. The pupils must work out the rule for the sequence (adding 3, or doubling the number); then they must predict the next number. Explain the following: the rule for the sequence is the algorithm; the process by which they worked it out was logical reasoning.

Decomposition

Break it to make it

Decomposition is the process of breaking down a problem into smaller manageable parts. This process helps us solve complex problems and manage large projects.

It has many advantages. The process of problem-solving becomes manageable. Large problems are daunting, but a set of smaller, related tasks is much easier to work with. Decomposition also means that a team can work together on the overall task, with each member bringing their own insights, experience and skills.

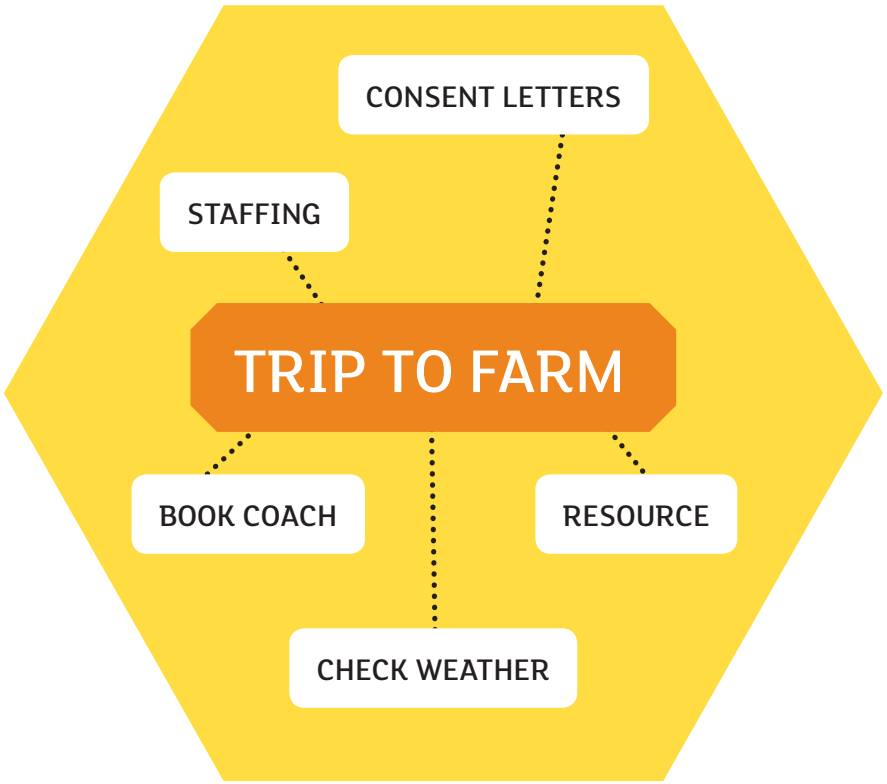
Decomposition in the real world

Breaking down problems into their smaller parts is not unique to computing. Decomposition is standard practice in engineering, design and project management. Software development is a complex process, so being able to split a large project into its component parts is essential. Consider the different elements that are combined to produce a complex program like PowerPoint. Computers themselves are similarly complex. For example, a laptop or a smartphone is made up of many components. These components are often made independently by specialist manufacturers, then assembled to form the finished product with everything working under the control of the operating system and applications.

How can we use decomposition in school?

You'll have used decomposition to tackle big projects at school already. Consider the examples that follow.

Delivering your school's curriculum is a good example. Typically a curriculum would be decomposed as years and areas of learning, then further into terms, units of work and individual lessons or activities. Notice how the project is undertaken by a team working together (your colleagues), and how important it is for the parts to integrate harmoniously. Organising events, such as a school play, a school trip or a school fair are all examples of projects in which decomposition is used.



Decomposition applied to a school trip

Decomposition used across the curriculum

It's likely that you and your pupils already use decomposition in many different ways across the curriculum. Here are just some examples.

- Labelling diagrams in Science or Geography to show the different parts of a plant, or the different nations which make up Europe.
- Planning the different parts of a story in English.
- Breaking down a problem in Maths.
- Planning a research project for any subject, or collaborating on a group presentation. Computers can be beneficial with this sort of collaborative group work, for example collaboration tools are available in Office 365 and other cloud-based software.

Where does decomposition fit in with the 5'E's?

The Using ICT Explore heading includes the requirement for pupils to 'investigate, make predictions and solve problems through interaction with digital tools'. This can include programming activities, and decomposition is an appropriate approach for pupils to use from Key Stage 2 onwards in this area.

The progression statements for Explore state that pupils should:

- at Level 3 'carry out and edit a series of instructions, make predictions and solve problems using a digital device or environment'; and
- at Level 4 'investigate and solve problems in a digital environment'.

As pupils plan programs to solve problems in a digital environment, encourage them to use decomposition by working out what the different parts of the program or system must do and thinking about how these are interrelated.

For example, a simple educational game needs:

- some way of generating questions;
- a way to check if the answer is right;
- some mechanism for recording progress such as a score; and
- some sort of user interface, which in turn might include graphics, animation, interactivity and sound effects.

Plan opportunities for pupils to work collaboratively on a software development project or on other projects in computing.

This could involve media work such as:

- animations or videos;
- shared online content such as a wiki; or
- a challenging programming project such as making a computer game or even a mobile phone app.

Classroom activities for decomposition

1

Plan a programming project

Plan a large-scale programming project, such as making a computer game in Scratch, through decomposition. Even for a relatively simple game the project would typically be decomposed as follows: planning, design, algorithms, coding, animation, graphics, sound, debugging and sharing. A project like this would lend itself to a collaborative, team-based approach, with development planned over a number of weeks.

2

Decompose a desktop

Remove the casing from an old desktop computer to reveal how computers are made from systems of smaller components connected together. It might be possible to disassemble some components further, though it could be easier to examine illustrations of their internal architecture.

3

Collaborative online project

Set up a collaborative project online. This might take the form of a multi-page wiki site. Pupils could, for example, take the broad topic of e-safety, decompose this into smaller parts and then work collaboratively to develop pages for their wiki, exploring each individual topic. The writing process for these pages might be decomposed further through planning, research, drafting, reviewing and publishing phases.

Abstraction

Abstraction means ignoring or hiding irrelevant details of a problem so you can focus on dealing with the important part.

For the American computer scientist, Jeanette Wing, who is credited with coining the term, abstraction lies at the heart of computational thinking:

'The abstraction process – deciding what details we need to highlight and what details we can ignore – underlies computational thinking.' Jeanette M Wing (2008) 'Computational thinking and thinking about computing'. http://rsta.royalsocietypublishing.org/content/roypta/366/1881/3717.full.pdf

When coders write code to carry out a task, their method may combine other methods or functions built into the programming language but they don't need to know how these functions were created.

While computer programmers find abstraction very useful, it's not something you need to explain to primary pupils when they're carrying out coding activities. It is, though, a powerful way of thinking about problems – so you may wish to introduce the concept in a simple way and not necessarily through computing lessons.

For example, in Maths, solving word problems often involves identifying the important information and setting out how to represent the problem in the more abstract language of arithmetic, for example by using the plus, minus and equals terms and symbols.

If we identify patterns, we can predict outcomes, make rules and solve general problems.

Patterns and generalisation

Making life easier

In computing, searching for a general approach to a class of problems is known as generalisation. If we identify patterns, we can predict outcomes, make rules and solve general problems.

Consider the following example. Pupils are learning about area by working out the area of a rectangle which is drawn on a grid of centimetre squares. While they could count the centimetre squares on the grid, a better method is to multiply the length by the width. Not only is this method faster, it also works for all rectangles, including those that are very small and those that are very big.

Of course, it does take time for the pupils to grasp this formula, but when they know it, this formula can be applied quickly and easily to similar problems that they face in the future. Time spent now is time saved later!

How are patterns and generalisation used in the primary curriculum?

Pupils have probably encountered the idea of generalising patterns in many areas of the primary curriculum already.

- Think about how they respond to nursery rhymes and stories. These contain repeated phrases and structures. Many children grow familiar with the repeated rhymes, rhythms and phrases of nursery rhymes even before they go to primary school. Later, they respond to the familiar narrative structures in traditional stories, such as in fairy tales.
- Songs and other pieces of music often contain patterns and melodies that children recognise and respond to.
- Pupils often work on simple maths problems that involve recognizing patterns and deducing generalised results.
- Pupils recognize patterns in spelling. These are often taught as rules, and exceptions to these general rules are also taught.

Classroom activities for patterns and generalisation

1

Number sequences

In computing lessons, encourage pupils to use a simple programming language such as Scratch to experiment with number patterns and sequences. Challenge them to work out a general program which can generate any linear number sequence.

2

Simpler, quicker, better

Teach pupils to find simpler or quicker ways to solve a problem or achieve a result. Invite them to create geometric patterns, working up to more complex 'crystal flowers' using turtle graphics commands in languages like Scratch, Logo or TouchDevelop. Some of the CCEA Using ICT tasks for Interactive Design could help with this. Emphasise how using repeating blocks of code is much more efficient than writing each command separately. Ask pupils to experiment with how changing one or two numbers in their program produces different shapes.

3

Graphic patterns

Teach pupils to use graphics software to create tessellating patterns to cover the screen. As they work on their patterns, ask the pupils to find faster ways to complete the pattern – copying and pasting groups of individual shapes is one easy way.

4

Sequencing in music

In Music lessons, teach pupils to use simple sequencing software in which patterns of beats are repeated, then allow them to use this to create rhythmic and effective compositions.

Writing software

In addition to the processes of computational thinking outlined above, there are numerous approaches that characterise computational thinking. It's worth helping pupils to develop such approaches to their work; if they do, they'll be much more successful in putting their thoughts into action.

Tinkering/Experimenting

Computer scientists love to play around with technology: to experiment and to explore. Aspects of learning a new programming language or exploring a new system are like the purposeful play that schools often encourage as an effective approach to learning in Foundation Stage classrooms.

Using open source is a great way to play with software and writing code. It's easy to take code written by someone else, to examine how it's constructed and then adapt it to your own purpose. This sharing-based approach is encouraged by programs such as Scratch. In fact, its name comes from the term 'scratching', which, in computing, refers to the process of reusing code – code that can be shared and reworked easily for other purposes and situations.

Rather than explain exactly how a new piece of software works, encourage pupils to play with it, sharing what they discover with each other. Also, set up opportunities for them to use code developed by others – by you, by their peers or from online sources. Use this code as the starting point for their own programming.

Creating

Creativity is at the heart of the programming process. Foster a spirit of creativity in your pupils by seeking out tasks that offer scope for their use of imagination and creativity, rather than simply have them program to find the right answer.

Teach them to reflect on the work they produce, thinking of the strengths and areas for development in their own and others' projects. It is now common practice for software developers to continually seek ways in which they can improve their software. Working with digital music, images, animation, virtual environments and 3D printing are some ways to foreground artistic creativity.

Debugging

A normal part of programming is writing code that initially contains flaws. These flaws, errors or faults are known as bugs, and a big part of writing code is their removal, which is known as debugging.

It's important for you to teach your pupils to think as programmers. Accordingly, they should take responsibility for thinking through their algorithms and code. They should also find and fix their mistakes. There are many areas of the curriculum in which similar processes are used. For example, in Language and Literacy pupils often draft, proofread and redraft their writing and, in Maths, pupils often check through their working.

Just as in Language and Literacy the teacher might ask pupils to proofread each other's writing, ask pupils to debug each other's code. Identifying and correcting mistakes independently is recognised as an important method of learning; debugging code – either a pupil working on their own code, or working on that of a peer – is an excellent way of doing this.

As the teacher, it's important for you to notice the bugs that creep in to your pupils' coding. Often these reveal misunderstanding that you can address directly, either with one or two pupils or with the group as a whole.

Persevering

It's worth reminding ourselves that computer programming is difficult. Yet that difficulty is part of its attraction. Overcoming the challenges along the way to produce code that works effectively brings the programmer tremendous satisfaction. A big part of overcoming those challenges goes beyond technical skills, such as knowing the right algorithms and understanding the language you're working in. A programmer must be willing and able to persevere with a task that's often difficult and sometimes frustrating. Qualities such as perseverance and resilience are vital.

Developing such qualities brings benefits in other areas of the curriculum too. As the work of the psychologist Carol Dweck shows, love of learning and resilience are qualities associated with virtually all high achievers. It is important for pupils to develop what Dweck terms a 'growth mindset', that is to believe that their abilities can grow through hard work and dedication. Meeting the challenges of programming and showing resilience through rewriting code and debugging are excellent ways to do just that.

Rather than allowing them to give up or to ask for your help when they encounter problems in their programming, teach pupils to adopt strategies for dealing with those problems. Typical strategies include identifying the exact nature of the problem, seeking a solution by using a search engine such as KidRex or Swiggle (or by using Bing or Google with the safe search mode locked), or by asking a classmate for help.

Collaborating

Software is developed collaboratively. Teams of programmers and others work together on a shared project. Set up activities that replicate this experience for your pupils. Group work is common in areas of the primary school curriculum; computing lessons – whether discrete, or as part of delivering another subject – should be no different.

'Pair programming' is one useful approach. Two programmers share a screen and keyboard as they work together to write software. One programmer usually adopts the role of driver (taking charge of the detail of the programming), whilst the other adopts the role of the navigator (taking charge of the 'big picture'). The two programmers swap roles regularly, so that each understands both the detail and the big picture.

Larger group work develops additional skills, with each pupil contributing some of their own particular talents or interests to a shared project. It's important, however, that all pupils develop their understanding of each part of the process. Do make sure you plan some sharing of roles or peer-tutoring throughout your activities.

Section 2
Programming

What is programming?

A program is a set of instructions for a computer, which is written in a language that it can understand. Programming is simply the action of designing and writing such instructions. At its simplest level, programming can be a process such as making a toy robot trace a square. At its most complex, it can be writing code to predict the weather.

There are two steps to writing a program:

1) Analyse the problem or system and devise a solution.

   This process uses logical reasoning, decomposition, abstraction and generalisation to design algorithms to solve the problem or model the system. (See Section 1 for further details of these processes.)

2) Express these ideas in a programming language.

   This is called coding. The set of instructions that make up the program is called code.

People study computer science to learn how to code. There's a wonderful sense of satisfaction when you achieve the goal of seeing a computer do just what you've asked because you wrote a precise set of instructions. Programming allows you to test ideas and have immediate feedback on whether something works or not.

Programming in schools

While it is possible to teach computational thinking without coding and vice versa, it's better to teach them together. Teaching computational thinking without allowing pupils to test their ideas by writing code on a computer is like teaching only scientific theories and principles without doing any experiments. Similarly, teaching programming without teaching the processes of computational thinking is like only doing experiments in science without teaching the principles that they exemplify.

It can be useful for pupils to analyse problems using terms such as algorithms and decomposition, and have repeated practical experience of writing computer programs in order to solve problems.

Programming activities for Key Stage 1 pupils could include looking at how simple algorithms are implemented as programs on digital devices. The phrase 'digital devices' includes tablets, laptop computers and programmable toys such as a Bee-Bot. It can be useful for pupils to be able to look at their algorithms, in whatever way they've recorded these, and their code side by side.

Pupils also should have the opportunity to create and debug their own programs as well as to predict what a program will do.

In Key Stage 2, pupils can move on to learning how to design and write programs that accomplish specific goals such as making a game in Scratch in which sprites interact. Other programs can include controlling or simulating physical systems, for example making and programming a robot. They can be taught to use sequence, selection and repetition in their programs as well as variables to store data (for example, using a counter to keep score in a Scratch game).

Pupils can also learn to use logical reasoning to detect and fix the errors in their programs. There are simple activities on the Barefoot Computing website; see the further resources listed at the end of this resource.

Here are some ideas for extended programming projects:

Key Stage 1

- Solve a maze using a floor/screen turtle.
- Create a simple animation in Scratch.

Key Stage 2

- Create a question and answer maths game.
- Create more complex computer games and animations in which there are more sophisticated interactions between sprites. For example, in Scratch, using the 'broadcast' and 'receive' commands to make characters in an animation talk to each other.

Using arrow cards to record algorithms for programmable toys.

How do you program a computer?

Programming a computer means writing code. The code is the set of instructions written in a language the computer can understand.

But don't despair: not all computer languages are highly complex and specialised. In fact, those that we teach and use at primary school are a halfway house. They're written in a computer language that is expressed in a version of English that we can understand, which then gets translated by the computer into a more pure computer language called machine code. Machine code is a set of instructions that can be run directly on the silicon chips of the machine. It is a language that the computer can respond to directly.

Programs are made up of precise instructions. When writing a line of code, there's no room for ambiguity or debate over meaning. We can only write code using the clearly defined vocabulary and grammar of the programming language, but typically these are words taken from English, so code is something that people can write and understand, but the computer can also follow.

Programming languages for primary school

There are hundreds of computer languages. While most of these are too complex for those beginning to learn programming, there are many that can be used effectively in the primary classroom. Many of these are also well resourced and some are backed by the support of online communities. Choose a language that you know already, or one that you'll find easy to learn.

When choosing a programming language, consider the following:

- Not all languages run on all computer systems.
- Select a language that suits your pupils. (There are computer languages that are readily accessible to primary pupils, such as Scratch. In most cases these have been written for pupils, or at least adapted to make them easier to learn.)
- Consider the learning resources that are available. Why not pick one that is supported by a good range? It's even better still if it has online support communities available, both for teachers and for learners.
- It is beneficial for the pupils to be able to continue working in the chosen language on their home computer. It's even better if they can continue work easily on the same project via the internet.
- Some say that some languages are better at fostering good programming habits than others. It's probably better to focus on the teaching than the language in that regard. At this early stage, good teaching – where computational thinking is taught alongside coding – will help to prevent pupils from developing bad coding habits.

The right language for the right Key Stage

The table below suggests which programming languages are suitable at which stage of primary school.

Key Stage	Language Type	Language
Foundation Stage	Device-specific	Bee-Bot
		Roamer-Too
KS1	Limited instruction	Scratch Jr
		Lightbot
KS2	Game programming	Kodu
	Block-based	Scratch, Hopscotch
	Text-based	Logo
		TouchDevelop

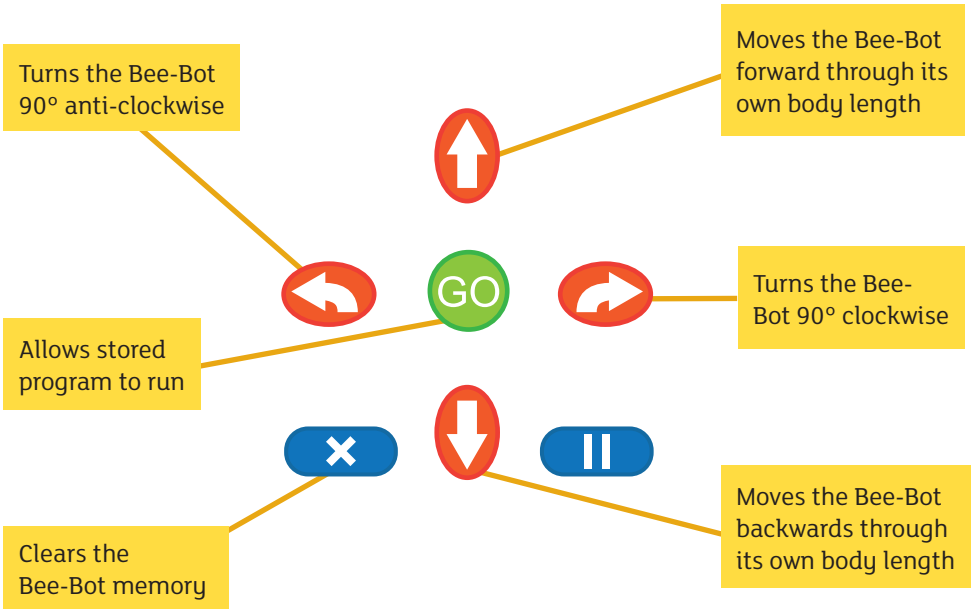
While it's tempting to let pupils explore several programming languages, it's more important that they develop a degree of fluency in a single one. Choose a general-purpose language: a medium in which they can accomplish functional tasks, solve problems and work creatively.

Programming a floor turtle

Rather than using computers, programming work at Foundation Stage and Key Stage 1 is more likely to use simple programmable toys.

Pupils learn the techniques of programming more quickly and easily when they use a simple language and a simple interface. Using a floor turtle, such as a Bee-Bot or a Roamer-Too, makes it easy for them to plan and check programs. Since these devices work on the floor, pupils can, quite literally, put themselves in the place of the device they're programming.

The programming language used by the Bee-Bot has five commands: forward, back, turn left, turn right and pause. All you need to do to program one is simply press the buttons in the desired order to build a sequence of commands, with new commands being added to the end of the sequence.



The Bee-Bot functions

© TSS Group Ltd

Pupils can enjoy a range of fun activities using a Bee-Bot, both for computing lessons and for work across the curriculum. Foundation pupils are likely to start programming the device one instruction at a time, but older children often become skilled at writing longer sequences.

When pupils are ready to move from programming floor turtles to programming on screen, why not use an on-screen simulation of a Bee-Bot? You can make, or adapt, one using Scratch Jnr.

Classroom activities

1. **Experiments with a turtle**
   Let very young pupils experiment with a floor turtle. As they play, encourage them to develop their understanding of the link between pressing buttons and running their program.

2. **Simple programming**
   The pupils plan a sequence of instructions to achieve a specific goal, such as moving the floor turtle from one 'flower' to another on an illustrated grid. Then the pupils must demonstrate logical reasoning by predicting what will happen when their program runs, and explain their thinking.

3. **More complex challenges**
   For more complex challenges, provide pupils with the code for a floor turtle's route from one place to another, including an error in the code. Ask the pupils to work out where the bug is in the code and then fix this, before testing out their code on the floor turtle.

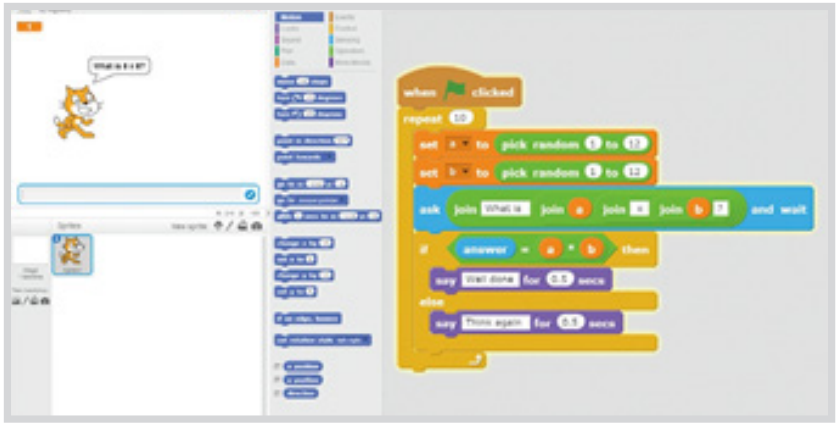
Programming movement on screen

Graphical programming toolkits make learning to code easy. In most of these, programs are developed by dragging or selecting blocks or icons which represent particular instructions in the programming language.

These can normally only fit together in ways that make sense, so the potential for errors in spelling or punctuation is minimised. In this way, the programmer focuses on the ideas of their algorithm and the intended outcome, rather than becoming bogged down in the words and the grammar of the programming language. Programming in this way usually means that pupils can become more active learners, who require much less support from their teacher.

Scratch

Devised by the prestigious Massachusetts Institute of Technology (MIT), Scratch is a versatile and free programming language that helps children learn programming. It has two versions: Scratch, which is aimed at students from the age of eight; and Scratch Jnr, which is aimed at children between the ages of five and seven. The programmer creates their own graphical objects, including the stage background where the action of the Scratch program takes place and moving objects, known as sprites, such as the characters in an animated story or a game.



Programming with Scratch

Each object can have one or more scripts, built using the blocks of the Scratch language (see the illustration above). To program an object in Scratch, drag the block you want from the different palettes then snap it into place within the other blocks on the screen. This forms a script. Scripts can run in parallel, or be set in motion by particular events.

Numerous other projects use Scratch as a starting point for their own platforms. For example, ScratchJr can be used in the form of an iPad app designed for young programmers at Foundation Stage and the start of Key Stage 1. The same sort of building block interface is used by Snap!, a language developed by the University of California at Berkeley, which allows even more complex programming ideas to be explored.

Scratch programmers can access a vibrant online community to download and share their projects globally. With such a network of support, pupils can learn much more about programming than what is required at primary school. Similarly, teachers can take advantage of help, support and high quality resources from Scratch's educator community. Scratch is available as both a web-based editor or as a standalone desktop application. You can move files between online and offline versions easily.

Kodu

Kodu was developed by Microsoft. It is visual programming language for creating simple, interactive 3D games. Each object in the Kodu game world can have its own program. Kodu programs are 'event driven'. This means that are made up of sets of commands set up in a 'when [this happens], do [that]' format. Actions are triggered when things happen, such as a key being pressed, one object hitting another or the score reaching a certain level.

As with Scratch, programmers can share their programs – in this case, games – with others in the Kodu community. This facilitates informal and independent learning. Again, like Scratch, pupils may download and modify games created by others. This can be an effective way to learn programming, which can encourage pupils to develop games with a strong sense of audience and purpose.

Classroom activities

1 **Scratch animations**
Pupils create simple scripted animation using Scratch. Each animation could have at least two programmed characters, who help to act out a story. Designing the algorithm for this sort of program is similar to storyboarding in video work.

2 **Kodu games**
First pupils play a selection of games from those on the Kodu community site. Then they are set the task of developing their own game. As a starting point, you might ask them to create a game in which Kodu (the player's avatar) is guided through a hostile landscape, where he encounters enemies.

Real programming

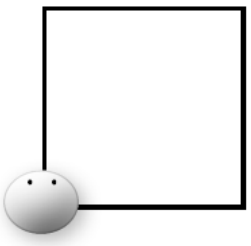
In universities and in industry, programming is done using text-based languages. Programmers write commands from the programming language using a keyboard.

In the past, text-based programming has proved a stumbling block for children who are learning to code, in primary school you needn't rush into text-based programming or feel obliged to include it in your curriculum. If, however, you or your colleagues feel confident in this area, it's worth considering starting a programming club in which pupils work on text-based programming. Suitable text-based programming languages for primary school pupils include Logo (there are ideas for using this within CCEA Using ICT tasks) and TouchDevelop.

Logo

Developed by Seymour Papert and others at MIT as an introductory programming language, Logo is best known for its use of 'turtle graphics'. This is a way of creating images in which a 'turtle' (either a robot turtle or a screen turtle) is given instructions for drawing a shape. The following is a simple example:

```
REPEAT 4 [
      FORWARD 100
      RIGHT 90 ]
```



Papert viewed Logo as a tool for children to think with. In Logo programming, more sophisticated programs are formed by 'teaching' the computer new words. These are called procedures. For example, you can teach the turtle how to draw a square of a given size using text understood by Logo (see below). After you have defined the procedure 'square', every time you type this word the turtle will draw a square. For example:

```
TO SQUARE :SIDE
      REPEAT 4 [
            FORWARD :SIDE
            RIGHT 90 ]
      END
SQUARE 50
```

TouchDevelop

TouchDevelop is Microsoft's interactive development environment and a visual programming language. It is used to develop apps for devices such as mobile phones and for tablets that use touch screen technology.

The physical process of typing on a phone or a tablet can be cumbersome, so learning to code on such devices can be problematic.

Although TouchDevelop is text-based, it is less difficult than many other text-based programming languages. Its programs are created by choosing commands from the options displayed in a menu system rather than by typing code directly. In this way, TouchDevelop might be considered a halfway house between graphical and text-based programming.

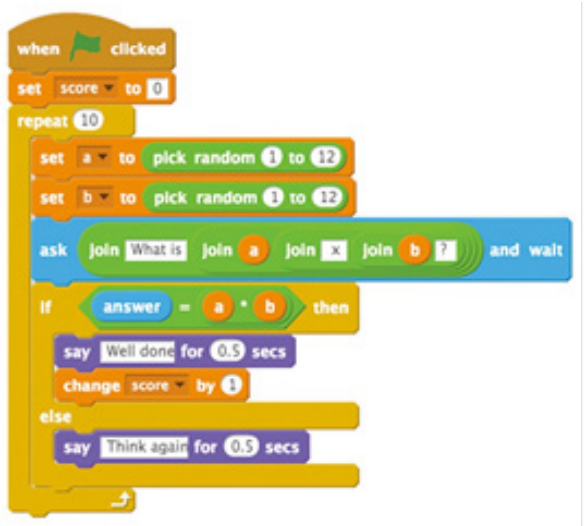
What's inside a program?

The details vary from language to language, but each one contains some of the same structures and ideas. Programmers use these over and over again – in different languages and to tackle different problems. The following is a summary of the structures.

- Sequence: running instructions in order (see p41);
- Selection: running one set of instructions or another, depending on what happens (see p43);
- Repetition: running some instructions several times (see p45); and
- Variables: a way of storing and retrieving data from the computer's memory (see p49).

It's important that pupils learn these concepts as they progress.

This Scratch script below shows sequence, selection, repetition and variables. Before we consider them in detail, can you work out which is which?



Identify the elements in this Scratch script

Sequence

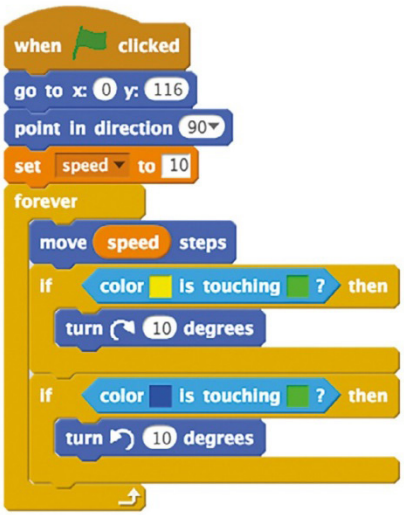
Programs are sequences of instructions. For example, the sequences of instructions in programs for floor turtles are built up as the stored sequence of button presses for what the turtle should do.

The instructions – as they would be for any program – are precise and unambiguous. The floor turtle will take each of the instructions (in the form of the stored button presses) and convert that instruction into a signal for the motors driving its wheels.

As they start to program, pupils might simply type a single instruction at a time, clearing the memory after each. Yet, as they gain experience as programmers, or need to solve a problem more speedily, their sequences grow in complexity. For example,

Forward
Forward
Forward
Turn left
Forward
Forward

When working with Scratch, pupils' early programs are likely to be made up of simple sequences of instructions too. These instructions must be precise, unambiguous and correctly ordered. In creating algorithms, pupils should have worked out the exact order in which to put the steps in order to complete the task.



Identify the elements in this Scratch script

Classroom activities

1. **Problem solving with a turtle**
   Give pupils progressively more complex problems to solve with a floor turtle. Ask them to plan their algorithm for solving these problems, then create single programs on the floor turtle.

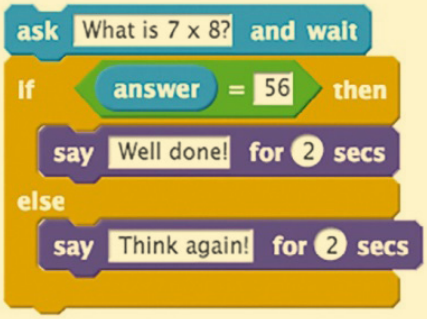
2. **Scratch remixed**
   Assign pupils existing projects from Scratch (see Further resources on page 55). Invite them to modify these projects by changing the code and observing the ways in which their modifications affect the program.

3. **Scratch animations**
   Using Scratch, pupils design, plan and code scripted animations. If possible, they should use a timeline or a storyboard to devise their algorithm before converting it into instructions for the sprites (characters) in Scratch.

Selection

Selection is the programming structure through which a computer executes one or other set of instructions according to whether a particular condition is met or not.

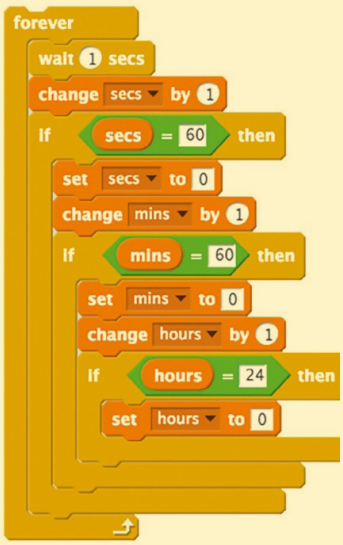
This ability to do different things depending on what happens in the computer as the program is run or out in the real world lies at the heart of what makes programming such a powerful tool.In Scratch, as with many other languages, we can build selection into our sequence of instructions. This allows the computer to follow different instructions according to whether a condition is met or not. The example below shows how this works in practice.



Selection being used in Scratch

A simple selection instruction lies at the heart of many educational games: if the answer is right then give a reward, else say the answer is wrong. (See the Scratch script for the times tables game later in this section.)

Selection statements can be 'nested' inside one another. This allows more complex sets of conditions to be used to decide what happens in a program.



Nested selection statements in a clock program

Look at how some 'if' blocks are inside others in this script written in Scratch. It is to model a clock in Scratch. Note how the script also uses repetition and three variables for the seconds, minutes and hours.

When designing a game in Kodu, selection is also vital. A set of conditions control an object's behaviour in a game. For example, WHEN you press the left arrow, the object moves left. Similarly, interaction with other objects, variables and environments in Kodu are programmed as a set of WHEN … DO … conditions. For example, WHEN I bump the apple DO eat it AND add 2 points to score.

Selection statements in Kodu

Classroom activities

1

Scratch quiz

Pupils use Scratch to design simple question and answer games. Teach them to devise the overall algorithm for their game before coding. Then encourage them to develop the user interface, making it more engaging than just a cat asking questions. Teach them to have a target audience in mind.

2

Kodu games

Pupils devise simple games using Kodu. Encourage them to experiment with the different conditions that a character in Kodu can respond to in its event-driven programming. Teach pupils consider how they might use such conditions when developing their own games. Allow them sufficient time to code and to redraft their code. Encourage them to think carefully about the algorithms they design – that is, the rules of their game. Barefoot computing has further, easy to follow lesson ideas for this activity including a link to a helpful introduction to Kodu video.
See www.barefootcas.org.uk/ for further details.

Repetition

In programming, repetition is when the program repeats the execution of certain commands.

Such an occurrence is also referred to as a loop, since the computer keeps looping through the commands one at a time as they're carried out. Using repetition makes a long sequence of instructions shorter and, usually, easier to understand.

Writing code that uses repetition typically involves noticing that some of the instructions you want the computer to follow are very similar or the same. It therefore draws on the computational thinking process of generalisation (see pages 24–26).

Consider the following two examples of writing a program to make a square.

Example 1:
A program for a Bee-Bot
(forward, left, forward, left, forward, left, forward, left).

Example 2:
As you can see, for each side we first move forward, then turn left. On a Roamer-Too or a Pro-Bot, we could use the repeat command to simplify the coding by using the built-in repeat command, replacing this code with, for example: **repeat 4 [forward, left].**

The same rules apply in Logo, which is the language that the Roamer-Too and Pro-Bot programming device-specific languages are derived from.

Compare the next two examples. Both are to draw equilateral triangles. The second uses repetition; the first does not.

1) Drawing a triangle (without repetition)

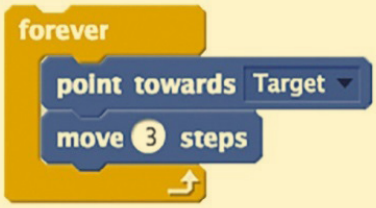
```
FORWARD 100
LEFT 120
FORWARD 100
LEFT 120
FORWARD 100
LEFT 120
```

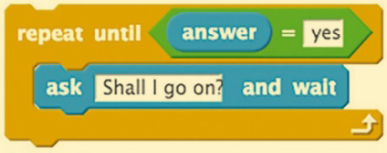
2) Drawing a triangle (using repetition):

```
REPEAT 3 [FORWARD 100
LEFT 120]
```

Notice how repetition reduces the amount of typing and how it makes the program reflect the underlying algorithm more clearly. In these examples, the repeated code is run a certain number of times. It's also possible to repeat code continually. This is useful in real world systems, for example in a control program for a digital thermostat. The program checks room temperature continually, and sends a signal to turn on the heating when it drops below a certain value. Such techniques are also common in games programming.

Consider, for example, the following Scratch code which makes a sprite continually chase another around the screen:



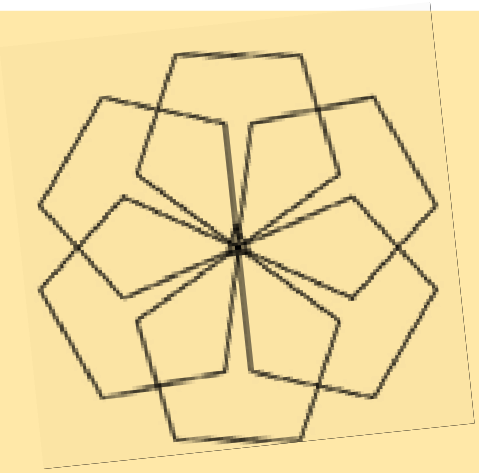


Repetition can be combined with selection. In this way a repeating block of code runs as many times as necessary until a condition is met. Look at the following example, which is a fragment in Scratch:

One repeating block can be nested inside another. 'Crystal flower' programs written in Logo use this idea. Consider the following example:

```
REPEAT 6 [
REPEAT 5 [
FORWARD 100
LEFT 72 ]
LEFT 60 ]
```

draws



Classroom activities

1

Fish tanks

Pupils produce a fish tank animation in Scratch. They use simple repetition commands to make each sprite independently with its own set of repeating motion instructions. They can add additional complexity by including some selection commands to alter the sprites' behaviour when they touch. See Scratch 2.0 Fishtank Game tutorial: www.youtube.com/watch?v=-qTZ5bFEdC8

2

Crystal flowers

Pupils experiment with crystal flower programs. They may use Scratch, Logo or other languages that support turtle graphics. They investigate the effect of changing the number of times a loop repeats as well as the effect of changing the parameters for the commands inside the loop. This activity offers plenty of scope to make links between computing and cultural education. See http://barefootcas.org.uk/ for an example of this in Scratch and Blackcat Logo helpsheets on the Primary Using ICT area of the CCEA website: http://ccea.org.uk/

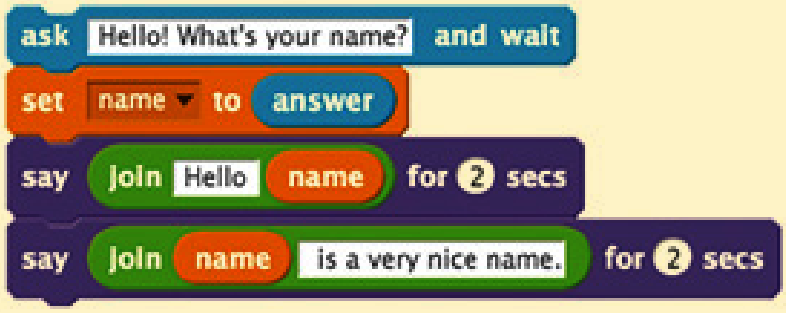
Variables

A variable is a simple way of storing a piece of information in the computer's memory while the program is running, and retrieving that information later.

For pupils in primary school, variables are a sophisticated concept. So if you want to introduce variables to your pupils, it's a good idea to show them plenty of examples to help them understand. A classic example, which pupils are likely to be familiar with from computer games, is score. If you were writing code to make a computer game, you would most likely make a variable called 'score'. This variable would store information about the points gained during a game. When, for example, the character collects a piece of treasure, you instruct the program to increase the variable 'score' by one. As points are collected, the variable keeps changing.

You can use variables to store data input by the person using your program and then refer to that data later on.

The following example, in which the user's name is remembered, comes from code written in Scratch.

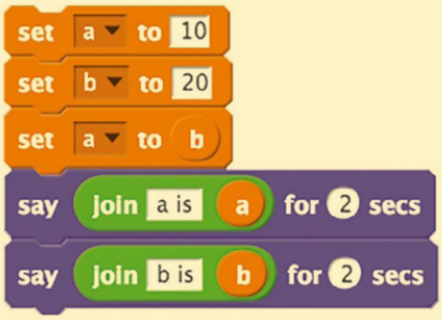


Name as a variable in Scratch

Here, 'name' is a variable. The computer stores whatever the user types in, then uses it twice in Scratch's response; 'answer' is a temporary variable used by Scratch to store for the time being whatever the user types in. As you can see from this example, variables can store text as well as numbers. You can store other types of data in variables too. This depends to a degree on the programming language you're working with.

Inexperienced programmers sometimes find it challenging to grasp the idea that the contents of the 'box' still remain there after the variable is used.

Read the following code and work out what will be displayed on the screen:

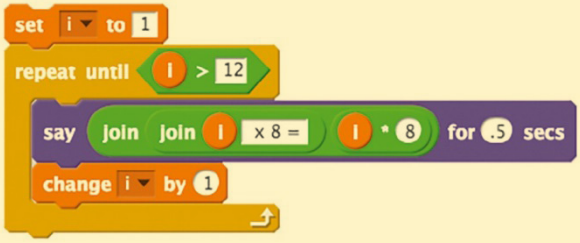


Using variables in Scratch

You should see 'a is 20' followed by 'b is 20'. Try it!

One way to make good use of variables in a program is to use them as an iterator. This counts the number of times a repeating loop has been completed. Simply set a counter to zero or one at the start of the loop, then add one each time the loop is completed.

For example, the following script in Scratch calculates the eight times tables.



Using an iterator

An iterator can also work with words and sentences (or strings as they're called in computing) one letter at a time, or through lists of data one item at a time. Be careful with the start and the finish. Beginning or ending too late or too soon is a common mistake when setting up iterators.

Classroom activities

1

Mystery function machine

Pupils create a mystery function machine in Scratch. This accepts an input, stores it in a variable, then uses mathematical operators to produce an output which is shown on screen. With the display set to full screen, pupils can challenge one another (and you) to work out what the program does by trying different inputs.

2

Games

Pupils use variables in their Scratch games programs. They use scoring to reward a player for achieving particular goals (for example, collecting apples), and they set a time limit.

Debugging is a vital part of writing code, but doing it can be more time consuming than writing the code in the first place!

Debugging: Can we fix it?

‘Bugs’ is the term given to mistakes in code and in algorithms. The process of searching for and repairing bugs is known as ‘debugging’.

Debugging is a vital part of writing code, but doing it can be more time consuming than writing the code in the first place! Bear this in mind as you plan coding activities. And remember that while debugging successfully and completing a working program brings great satisfaction, poring over code that refuses to work can cause great frustration. Be prepared to manage this in class.

It’s useful for pupils in Key Stage 2 to use logical reasoning to find and fix errors in algorithms and programs. Pupils can correct their code and explain what went wrong and how they fixed it.

In programming classes, pupils writing a program for a particular purpose might want you or others to repair their programs. While your first instinct might be to help, please remember that the objective is not so much to have pupils create a working program, as for them to learn how to program. Debugging skills are a crucial part of that.

How do you debug?

It’s good practice to provide a robust, general set of debugging strategies which pupils can use for any programming activity.

Logical reasoning should underpin debugging. This is apparent in the approach of the Barefoot Computing team. They suggest a logical debugging sequence of four steps.

1. Predict what should happen.
2. Find out exactly what happens.
3. Work out where something has gone wrong.
4. Fix it.

One way to help predict what should happen is to get pupils to explain their algorithm and code to someone else. In doing so, it’s quite likely that they’ll spot a flaw in the way they’re thinking about the problem or in the way they’ve coded the solution.

In finding out exactly what happens, it can be useful to work through the code, line by line. Seymour Papert described this as ‘playing turtle’. So, in a turtle graphics program in Logo (or similar) pupils could act out the role of the turtle, walking and turning as they follow the commands in the language.

In working out where something has gone wrong, encourage pupils to look back at their algorithms before they look at their code. Before they can begin to fix bugs, they need to establish whether it was flaw in their thinking or with the way they've implemented that as code.

Some programming environments allow you to step through code one line at a time – you can do this in Scratch by adding (wait until [space] pressed) blocks in liberally. Scratch will default to showing where sprites are and the contents of any variables as it runs through code, which can also be useful in helping to work out exactly what caused the problem.

Debugging is a great opportunity for pupils to learn from their mistakes and to get better at programming.

Classroom activities

1. **Fixing pre-bugged programs**
While pupils will probably make plenty of authentic errors in their own code, it's also worth training them to debug by giving them other programs to fix. This can help them learn strategies for debugging. It also helps you assess their skills in logical reasoning as well as their programming knowledge. Create some programs containing deliberate mistakes, perhaps using a range of semantic errors (errors in the logic of the code). Set pupils the challenge of finding and fixing these.

2. **Peer Debugging**
Pupils debug each other's programs. One way to manage this is that pupils work on their own programs for the first part of the lesson then take over their partner's project. Next they complete this project then they debug it for their partner.

3. **Deliberate peer errors**
Pairs of pupils to write code containing deliberate mistakes. They challenge to their partner to discover and repair the errors in the code.

Further reading and resources

General

The following sites contain a wealth of practical and engaging ideas, information and approaches.

Barefoot computing

http://barefootcas.org.uk/

The Barefoot Programme supports primary school teachers, equipping them with the confidence, knowledge, skills and resources to teach computing. Supported by the Department for Education, and endorsed by the Chartered Institute for IT, this site includes free high-quality resources and lesson plans.

Resources to support all of the areas in this guide – both on computational thinking and programming – are available via the barefoot site and have been referenced to the Northern Ireland Curriculum.

BBC Bitesize

http://www.bbc.co.uk/education

The Bitesize site has a range of learner guides and programmes to support primary school pupils on all aspects of computing. Follow the links from the main site to KS2 and then to Computing.

Quickstart

http://primary.quickstartcomputing.org/

This website covers all of the material in this guide and offers additional material, such as videos and links to further resources. (The content of this guide has been adapted from the QuickStart Computing toolkit for primary pupils.)

Wider reading

Miles Berry

http://milesberry.net/
The personal website of Miles Berry, principal lecturer for Computing Education at the University of Roehampton and the author of the QuickStart Computing toolkit, contains links to many of his articles and other publications on computing and education.

Topics

Computational thinking

Berry, M, 'Computational Thinking in Primary Schools' (2014):
http://milesberry.net/2014/03/computational-thinking-in-primary-schools/

Computer Science Teachers Association, 'CSTA Computational Thinking Task Force' and 'Computational Thinking Resources':
http://csta.acm.org/Curriculum/sub/CompThinking.html

Computing at School, 'Computational Thinking':
http://community.computingatschool.org.uk/resources/252

Curzon, P, Dorling, M, Ng, T, Selby, C and Woollard, J, 'Developing Computational Thinking in the Classroom: A Framework' (Computing at School, 2014), available at:
http://community.computingatschool.org.uk/files/3517/original.pdf

Google for Education, 'Exploring Computational Thinking':
www.google.com/edu/computational-thinking/index.html

Wing, J, 'Computational thinking and thinking about computing' (The Royal Society, 2008):
http://rsta.royalsocietypublishing.org/content/366/1881/3717.full.pdf+html

Logical reasoning

Computer Science for Fun, 'The Magic of Computer Science', available at:
www.cs4fn.org/magic/

Computer Science Unplugged, 'Databases Unplugged', available at:
http://csunplugged.org/databases

McOwan, P and Curzon, P (Queen Mary University of London), with support from EPSRC and Google, 'Computer Science Activities with a Sense of Fun', available at:
www.cs4fn.org/teachers/activities/braininabag/braininabag.pdf

Decomposition

Basecamp (professional project management software) can be used by teachers with their class (free), available at:
https://basecamp.com/teachers

Gadget Teardowns, available at:
www.ifixit.com/Teardown

NRICH, 'Planning a School Trip', available at:
http://nrich.maths.org/6969

Project Management Institute Educational Foundation, 'Project Management Toolkit for Youth', available at:
http://pmief.org/learning-resources/learning-resources-library/project- management-toolkit-for-youth

Computational thinking approaches

DevArt: Art Made with Code, available at:
https://devart.withgoogle.com/

Education Endowment Foundation toolkit, available at:
http://educationendowmentfoundation.org.uk/toolkit/

Papert, S and Harel, I, 'Situating Constructionism' (Ablex Publishing Corporation, 1991):
www.papert.org/articles/SituatingConstructionism.html

CAS Chair, Prof Simon Peyton Jones' explanation of some of the computer science that forms the basis for a computing curriculum:
http://community.computingatschool.org.uk/resources/2936

Code.org for activities and resources:
http://code.org/educate

Rushkoff, D, 'Program or be Programmed: Ten Commands for a Digital Age' (OR Books, 2010)

Programming a moveable device

Bee-Bot and Roamer-Too simulator activities:
http://scratch.mit.edu/projects/19799927/

LightbotTM:
http://lightbot.com/

Graphical programming

The following are accessible and well-resourced languages aimed at primary pupils:
Scratch: http://scratch.mit.edu/
ScratchJr: www.scratchjr.org/

Using Scratch:

Armoni, M and Ben-Ari, M, 'Computer science concepts in Scratch':
http://stwww.weizmann.ac.il/g-cs/scratch/scratch_en.html

Berry, M, 'Scratch across the curriculum':
http://milesberry.net/2012/06/scratch-across-the-curriculum/

Scratch project to remix: Analogue clock by Miles Berry on Scratch:
http://scratch.mit.edu/projects/28742256/#editor

Creative Computing, 'An Introductory Computing Curriculum Using Scratch':
scratched.gse.harvard.edu/guide/

The following graphical programming languages may also be of interest:
Snap!: http://snap.berkeley.edu/
Kodu: www.kodugamelab.com/

Text-based programming

Logo: www.calormen.com/jslogo/ **and elsewhere, including Blackcat Logo, available through C2K 'MySchool'.**

Papert, S, 'Mindstorms: Children, Computers, and Powerful Ideas'
(Basic Books Inc., 1980):
http://dl.acm.org/citation.cfm?id=1095592

TouchDevelop interactive tutorials for Hour of CodeTM:
www.touchdevelop.com/hourofcode2

TouchDevelop from Microsoft Research:
www.touchdevelop.com/

Sequence

Animation 14: UK Schools Computer Animation Competition (Key Stage 2):
http://animation14.cs.manchester.ac.uk/gallery/winners/KS2/

Selection

Papert, S, 'Does Easy Do It? Children, Games, and Learning':
www.papert.org/articles/Doeseasydoit.html

Repetition

Digital Schoolhouse dance scripts:
http://resources.digitalschoolhouse.org.uk/key-stage-2-ages-7-10/218-scratch-teaching-dance

Scratch 2.0 Fishtank Game tutorial:
www.youtube.com/watch?v=-qTZ5bFEdC8

Variables

Bagge, P, 'Text Adventure Game' for Scratch:
http://code-it.co.uk/text_adventure_game/

Binary search jigsaw and solution by Miles Berry:
http://scratch.mit.edu/projects/20255402/ and http://scratch.mit.edu/projects/28907496/

Notes and tutorial on variables in Scratch:
http://wiki.scratch.mit.edu/wiki/Variable and http://wiki.scratch.mit.edu/wiki/Variables_ Tutorial

Debugging

Debugging challenges from Switched on Computing:
http://scratch.mit.edu/studios/306100/

Books

Dweck, C, 'Mindset: How You Can Fulfil Your Potential' (Robinson, 2012)

Cormen, T, 'Algorithms Unlocked' (MIT Press, 2013)

Kelly, J, Kodu for Kids (Que Publishing, 2013)

Laurillard, D, 'Teaching as a Design Science: Building Pedagogical Patterns for Learning and Technology' (Routledge, 2012)

Steiner, C, 'Automate This: How Algorithms Came to Rule Our World' (Portfolio Penguin, 2013)

Glossary

Abstraction (process): the act of selecting and capturing relevant information about a thing, a system or a problem.

Acceptable Use Policy (AUP): An Acceptable Use Policy comprises a set of rules applied by the owner/manager of a network, website or large computer system that defines the ways in which the network, site or system may be used.

Algorithm: An unambiguous set of rules or a precise step by step guide to solve a problem or achieve a particular objective.

Command: An instruction for the computer to execute, written in a particular programming language.

Computational thinking: Thinking about systems or problems in a way that allows computer systems to be used to model or solve these.

Computer networks: The computers and the connecting hardware (wifi access points, cables, fibres, switches and routers) that make it possible to transfer data using an agreed method ('protocol').

Data: A structured set of numbers, possibly representing digitised text, images, sound or video, which can be processed or transmitted by a computer, also used for numerical (quantitative) information.

Debug: To fix the errors in a program.

Decomposing/ decomposition: The process through which problems or systems are broken down into their component parts, each of which may then be considered separately.

E-safety: Used to describe behaviours and policies intended to minimise the risks to a user of using digital technology, particularly the internet.

Generalisation: A computational thinking process in which general solutions or models are preferred to or derived from particular cases.

Hardware: The physical systems and components of digital devices; see also software.

Input: Data provided to a computer system, such as via a keyboard, mouse, microphone, camera or physical sensors.

Interface: The boundary between one system and another – often used to describe how a person interacts with a computer.

Loop: A block of code repeated automatically under the program's control.

Open source software: Software in which the source code is made available for others to study, and typically adapt, usually with few if any restrictions.

Operating system: The programs on a computer which deal with internal management of memory, input/output, security and so on, such as Windows 8 or iOS.

Output: The information produced by a computer system for its user, typically on a screen, through speakers or on a printer, but possibly through the control of motors in physical systems.

Packets of data: A small set of numbers that get transmitted together via the internet, typically enough for 1000 or 1500 characters.

Program: A stored set of instructions encoded in a language understood by the computer that does some form of computation, processing input and/or stored data to generate output.

Programming (or coding) is the term given to the writing of programs.

Programmable toys: Robots designed for children to use, accepting input, storing short sequences of simple instructions and moving according to this stored program.

Repetition: Executing a section of computer code a number of times as part of the program.

Script: A computer program typically executed one line at a time through an interpreter, such as the instructions for a Scratch character.

Selection: A programming construct in which one section of code or another is executed depending on whether a particular condition is met.

Sequence: To place program instructions in order, with each executed one after the other.

Simulation: Using a computer to model the state and behaviour of real-world (or imaginary) systems, including physical or social systems; an integral part of most computer games.

Software: computer programs such as Office programs, web browsers, games and the computer operating system. The term also applies to apps running on mobile devices and to web-based services.

Sprite: A computer graphics object that can be controlled (programmed) independently of other objects or the background.

Variables: A way in which computer programs can store, retrieve or change data, such as a score, the time left, or the user's name.

Part Two – Post Primary

Content adapted from:
Woodman, A., *QuickStart Computing: a CPD toolkit for secondary teachers*, Swindon: BCS.

A catalogue record for this title is available from the British Library.
ISBN: 978-1-47184-807-0

http://www.quickstartcomputing.org/secondary/848070_QS_Comput_SO.pdf

Introduction

In the Northern Ireland Curriculum at Key Stage 3 (KS3), ICT (Information and Communication Technologies) is not a discrete Area of Learning. Rather, Using ICT is a cross-curricular skill. It is to be developed and assessed across all subject strands in the curriculum alongside the cross-curricular skills of Communication and Using Maths.

Computing is a new subject in the English National Curriculum, developed in order to draw together strands of Computer Science, Information Technology and Digital Literacy for pupils. While not a discrete Area of Learning in the Northern Ireland Curriculum, the knowledge and teaching approaches of computing align strongly with the types of classroom activities that teachers in Northern Ireland already use. Such activities are to develop and assess Using ICT skills as set out in the Levels of Progression, and with aspects of the Thinking Skills and Personal Capabilities (TSPC) framework.

A high-quality computing education 'equips pupils to use computational thinking and creativity to understand and change the world' (Department for Education, 2013). This guide supports teachers from a broad range of backgrounds who want to understand the best ways to deliver computing and computational thinking within the context of their own subject area. It also aims to provide school leadership with a starting point as they seek to understand the best ways to develop computing and computational skills across the curriculum.

As with our guide to computing at primary school, this guide focuses on computational thinking and programming/coding. We hope that the information and support here help you plan Exploring Programming[2] activities that incorporate the Explore, Express and Evaluate elements of Using ICT: to 'investigate, make predictions and solve problems through interaction with digital tools'. While we include some detail on the key concepts of computing, much of this guide emphasises computational thinking and strategies and lesson ideas that can be used in the classroom.

Computational thinking as defined in this guide also overlaps extensively with the Northern Ireland Curriculum requirements to develop pupils' Thinking Skills and Personal Capabilities. We define TSPC as 'tools that help children to go beyond the acquisition of knowledge to search for meaning, apply ideas, analyse patterns and relationships, create and design something new and monitor and evaluate their progress'.

[2] *Exploring Programming is the term we use to describe the coding and programming activities typically seen in post-primary classrooms. We have created non-statutory guidance for this area to help teachers plan and assess pupils.*

The three main strands within computing

The Royal Society identifies three distinct strands within computing, each of which complements the others: Computer Science, Information Technology (IT) and Digital Literacy (see Figure 1). Each component is essential in preparing pupils to thrive in an increasingly digital world.

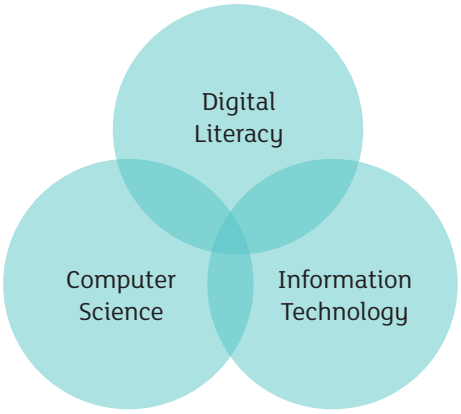


Figure 1: The complementary strands within computing

Computer Science is the scientific and practical study of computation: what can be computed, how to compute it, and how computation may be applied to the solution of problems.

Information Technology is concerned with how computers and telecommunications equipment work and how they may be applied to the storage, retrieval, transmission and manipulation of data.

Digital Literacy is the ability to effectively, responsibly, safely and critically navigate, evaluate and create digital artefacts using a range of digital technologies.

The creation of digital artefacts is integral to much of the learning of computing. Digital artefacts can take many forms, including digital images, computer programs, spreadsheets, 3D animations and this booklet.

Computing and the Northern Ireland Curriculum

The Northern Ireland Curriculum emphasises the development of skills and capabilities for lifelong learning and for contributing effectively to society. These whole curriculum skills and capabilities consist of the cross-curricular skills (including Using ICT) and Thinking Skills and Personal Capabilities.

Computational thinking is a core part of Computer Science. Its development through the rigorous and creative process of writing code helps develop pupils' problem-solving skills. At KS3, there is an increasing emphasis on 'audience and purpose' in all Using ICT activities, as pupils use a wide range of digital technologies to create products with a clear end user and purpose in mind. This focus on audience and purpose is also a key part of computer programming. It is important that an excessive focus on language skills and syntax is avoided. This could detract from the creation of practical solutions to real-world problems.

Exploring Programming has always been included in CCEA's non-statutory guidance materials for Using ICT. However, it has frequently been overlooked or seen as a niche topic best suited to the Science and Technology Area of Learning. This need not be the case. Indeed, much of this describes and illustrates how programming can be used across the curriculum as a fun way to explore various topics while developing a range of computing skills.

The requirements for Using ICT are set out under headings often described as the 5'E's. Computer programming activities link closely with Explore, Express and Evaluate. In Explore pupils 'investigate, make predictions and solve problems through interaction with digital tools'. In Express 'pupils create, develop, present and publish ideas and information responsibly using a range of digital media and manipulate a range of assets to produce multimedia products'. In Evaluate pupils 'talk about, review and make improvements to work, reflecting on the process and outcome... including safety, reliability and acceptability'. All these requirements match well with the process involved in high-quality computer programming activities.

Computational thinking

Computational thinking is central to computing. It is the process of recognising aspects of computation in the world that surrounds us and applying tools and techniques from computing to understand and reason about both natural and artificial systems and processes.

Computational thinking is a distinct way of thinking and working – one that can be applied in many contexts outside of computing. It allows pupils to tackle problems, to break them down into solvable chunks and to devise algorithms to solve them.

In summary, computational thinking involves:

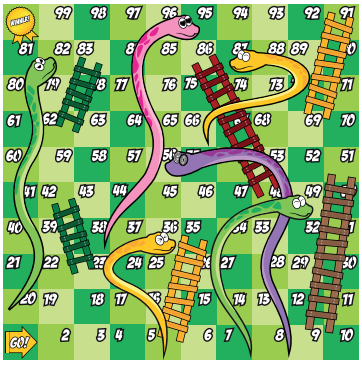
- decomposition;
- pattern recognition;
- abstraction;
- pattern generalisation; and
- algorithm design.

Example: Snakes and Ladders

To illustrate these concepts, let's look at how a Snakes and Ladders computer game might be made.

Decomposition is breaking a problem down into its components, each of which can be tackled individually and further decomposed. For example, Snakes and Ladders needs a model that captures:

- a board;
- snakes and ladders on the board;
- player counters;
- a six-sided die; and
- rules that describe when and how to move counters.



Pattern recognition is looking for similarities in the behaviours and states of the system you are trying to model. For example, you might specify in the game:

- that every time a counter meets the top of a snake it goes to the bottom of that snake; and
- to complete a game, the player's counter must land on square 100.

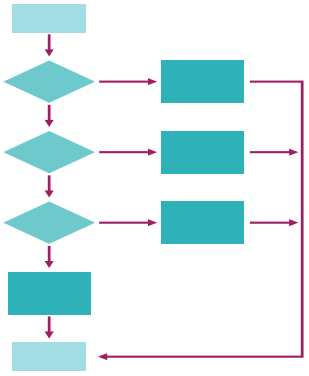
Abstraction helps you only use the detail absolutely necessary for the functioning of the system. Computational abstractions are models (often pieces of code) that include sufficient information to represent the computational aspects of a situation without describing absolutely everything. For example, Snakes and Ladders might model a snake as two sets of co-ordinates, ignoring the colour of the snake or the fact that a real snake would need to eat!

Pattern generalisation allows us to define concepts in their simplest form and to re-use the definition for all instances of that concept. For example, all the snakes in our game could be stored as just two sets of co-ordinates: one for the top and one for the bottom.

An **algorithm** is a precise method for solving a given problem. In this case, the method comprises the steps of rolling a die, moving a counter, ascending a ladder, changing turns, detecting when the game has finished, etc.

Algorithms are not just used by computers. For instance, the algorithm for 'repair a puncture on your bike' might be:

- take the wheel off the bicycle;
- remove the tyre from the wheel;
- remove the inner tube from the tyre;
- find the hole in the inner tube;
- patch the hole in the inner tube;
- put the inner tube back into the tyre;
- replace the tyre back on to the wheel; and
- put the wheel back on the bicycle.



As teachers, the majority of us are competent and confident users of technology in our own personal and professional lives. Few of us have sought to understand **how** our computers work, or how to program a computer. Many of us are unsure how to teach these things to our pupils.

Now, with help from the web, from new publications and resources, from online communities and from our colleagues (and pupils), it's time to give it a go.

Algorithms

An algorithm is a precise method for solving a given problem (for example, a recipe for baking a loaf of bread, a knitting pattern or instructions for constructing a flat-pack desk).

Some algorithms are written to be run on a computer, and some are meant to be carried out by a person. Programs contain algorithms. An algorithm can also take the form of steps written in structured English (pseudocode), or it can be expressed as a flowchart.

Algorithms help shape the world around us. We have algorithms to find the best route from A to B, algorithms that rank pages from a web search, algorithms that establish our credit rating and so on. Algorithms are everywhere!

Often, there are several possible algorithms to solve the same problem, which differ in complexity, efficiency or generality. These may just be different ways of arriving at the same result (for example, different procedures for adding together three-digit numbers). Where the problem is more complex, different algorithms may lead to different solutions (for example, different routes from A to B).

It is important that pupils understand there may be more than one way to solve a problem, and that some problems have more than one 'right' solution. This can be challenging, but it is an essential lesson in life. Pupils should be prepared to evaluate an algorithm using factors such as correctness (in the sense that it solves the problem) and speed, but also the quality of the solutions that it yields.

For example, to find the way out of a maze, one (simple but slow) algorithm might be to walk around at random until you find the exit. Another (more complicated) algorithm would involve remembering where you had been in order to avoid going down the same path twice. Another might be to keep your left hand on the wall and walk until you find the exit. Each algorithm gives a different solution to the same problem (a different route out of the maze). Other qualities might be taken into consideration: maybe one solution gets your feet wet and another gets them muddy.

Key algorithms

There are a number of tasks that are common to a wide range of programs. Among these are searching and sorting. It is important that pupils understand these and that key algorithms have been devised to carry them out. This will avoid excessive time spent trying to come up with solutions that are readily available. Searching and sorting are so common in programming that it is worth discussing them here. Exploring them also makes for good active learning activities and plenty of lesson resources are available online.

Searching

Linear and binary searches are the main searches pupils should learn.

A **linear search** starts at the beginning of a list and goes through every item until it finds the one you are looking for (or you come to the end of the list without finding the item). A real-life example of this might be looking for a picture of your cousin's wedding in a pile of unordered printed photographs. You keep flicking through until you find the photograph, or until you've looked at all the photographs and conclude that the picture isn't there.

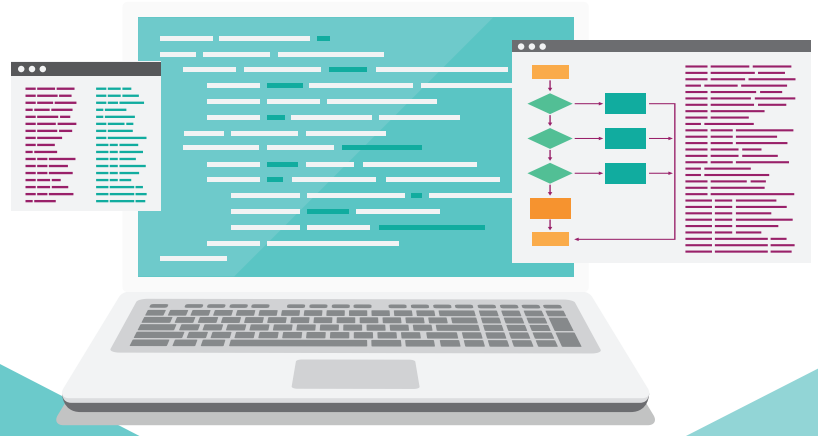
A **binary search** is much quicker, but only works if you have a list of items that have already been sorted in order. You start with the middle item. If you find it first time, well done! Otherwise, the item you are looking for will either be in the top or bottom half of the list. Go to the middle of that half and eliminate the other half of the list. Continue chopping the list in half until you have found the item – or worked out that the item you're looking for is not in the list.

You may find it useful to look at http://csunplugged.org/searching-algorithms which provides lots of non-computer-based searching activities.

Sorting

There are dozens of competing algorithms for sorting data. It is important that pupils know that there are lots of ways of achieving the same result, and that some methods are more efficient than others. Giving pupils physical sorting tasks, and asking them to record the algorithms they use, can be a useful way of introducing the subject.

There are some helpful examples at http://csunplugged.org/sorting-algorithms, which provide lots of non-computer-based sorting activities. There are even sorting dance videos. Why not conduct a search, typing 'sorting dance videos' and clicking on the videos tab of your search engine?



Using pseudocode and flowcharts in planning

When designing a solution to a problem, the process of computational thinking and algorithm design is key.

Coding in a specific language is not required straight away. In fact it can hinder the design process. To write in code requires variables and constants to be declared and all the syntax of the code – such as semi-colons, indentation and brackets – to be correct. This is time-consuming, and error-checking in the language will distract pupils from thinking about the quality of the solution itself. It may even deter pupils from looking at a range of possible solutions.

Pseudocode

A better solution to planning the program is to write it in pseudocode. Pseudocode is a mix of English and generic computer code. It allows pupils to determine how they want the program to work as well as what variables and functions/methods they need to include. It also helps them work through their logic and this should reduce the number of errors they will have to fix in the finished program.

Let's consider a simple example.

A pupil is designing a game in which the computer generates a random number and the user has to guess its value. The pupil has written the following algorithm for the game process.

1. Generate a random number between 1 and 50.
2. Ask the player to type in a number.
3. Compare the player's guess with the secret number.
4. If the numbers are the same, go to the next step. Otherwise tell them if they are too high or low and return to step 2.
5. Display a message telling the player they guessed the number correctly.

Representing this algorithm as pseudocode the pupil might write something like this:

```
Function Check_Guess
randomNumber = (1..50)
Get playerGuess from text box
If playerGuess = randomNumber Then
        Display "Correct"
Else If playerGuess > randomNumber Then
Display "Guess too High"
Else
Display "Guess too Low"
End
```

Flowcharts

Another option is to use a flowchart. A flowchart uses symbols and text to give a visual representation of a solution to a problem. A particular strength of a flowchart is that it uses arrows that show the flow of the logic. This works well in designs involving a lot of Boolean logic. (Boolean logic is a form of algebra in which all values are reduced to either TRUE or FALSE.)

There are a number of flowchart programs. Technology teachers usually have some experience in Logicator. However, pupils can create flowcharts in Microsoft Word or even on paper. Different symbols are used to represent each stage and the text in the symbols is the equivalent of pseudocode. Arrows indicate the flow through the program.

For the random number game above, the flowchart could look like Figure 2:

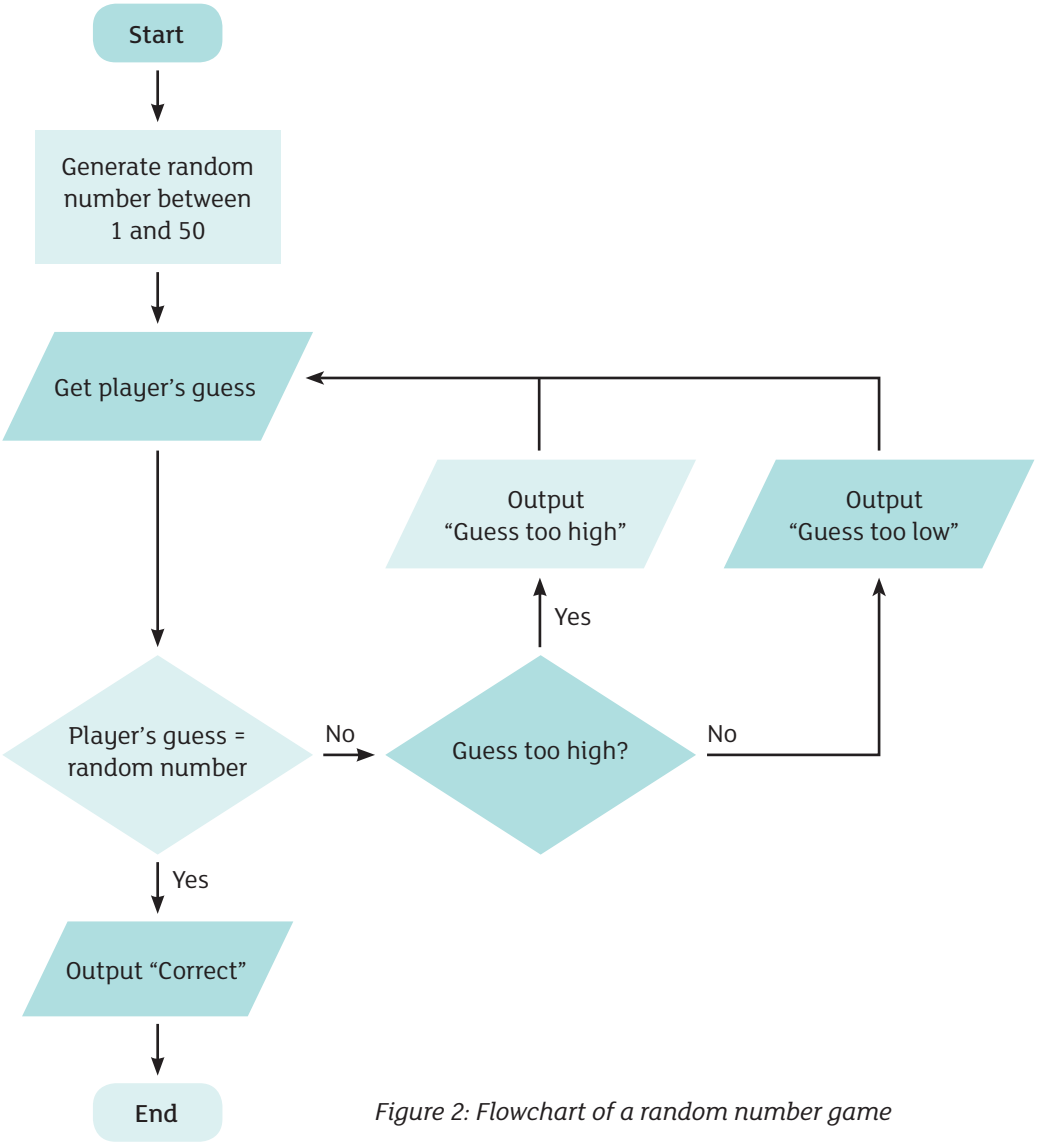


Figure 2: Flowchart of a random number game

Programming concepts

While this guide is not intended as a detailed manual to computer programming, there are a number of key concepts that need to be understood before pupils can tackle programming problems.

Sequence

Sequencing is putting the instructions in the correct order, for example asking the user how many sides a polygon should have before attempting to calculate the angles and draw it.

Selection

Selection is using conditions to control the flow of a program. For example:

“IF username = “Sam” and password = “j377y78”

THEN display welcome message

ELSE display error message”.

Repetition

This is the ability to execute a sequence of instructions many times until a certain condition has been met. For example:

“WHILE username incorrect or password incorrect DO ask for username and password”.

Boolean Logic

Boolean statements can have one of two values: TRUE or FALSE. Why is this important for pupils to understand? Truth values and Boolean logic are fundamental to how computer programs work. When pupils use IF or WHILE statements they will always use truth values and often need Boolean logic. For example:

“IF Health < 50 AND Character is touching enemy THEN Game Over”.

It is useful to encourage pupils to plan their programs and break down the logic involved using pseudocode or flowcharts.

Other Concepts

Computers function as **input**, **process** and **output** systems. Data is inputted, computation is performed and an output response is given. For example, when you type in your username and password, they are checked against stored values, and the output is either a failure or success message. Pupils should be aware of a range of input and output devices, what they are used for and, at a very basic level, how they provide an interface between the rich, analogue real world and the digital domain of the computer.

Planning

A Cross-Curricular Skill

Using ICT is a cross-curricular skill in the Northern Ireland Curriculum so computing, as a powerful interdisciplinary subject, potentially could be developed across every Area of Learning. This will be challenging for many subject specialist teachers and will require dedicated time in order to implement it. However, finding cross-curricular applications can be of real benefit, especially as they allow pupils to apply computational thinking across a range of subjects. There are benefits within each subject too. As pupils write programs based on key subject material, their depth of understanding of that subject increases.

A later section of this guide provides a range of possible cross-curricular programming project ideas. These can be used or adapted to suit individual subject needs/preferences. There are also many ideas online to support the development of computational thinking and programming skills. Many valuable activities arise naturally from within subject content, and applying computational thinking, or developing a programming project, is a valuable way of enhancing pupils' understanding of your subject.

Progression

Pupils arrive at KS3 with differing knowledge and experiences of computing, so early assessment and intervention may be needed. KS3 is a stepping stone for future qualifications and careers. Planning needs to cover all parts of the curriculum in sufficient depth so that pupils can make informed choices about their future. This includes the cross-curricular skills in addition to subject knowledge.

As pupils move through KS3, planning should take account of the fact that many computing concepts take time to master. Computational thinking and programming must be practised and key ideas, such as algorithmic problem solving, need to be revisited again and again. This requires a whole-school approach to co-ordinate when and where these skills will be acquired and developed. When designing an activity, it is important that you determine what prior knowledge is required and consider what skills need to be addressed or revised.

Pupils in your classes have a range of programming and digital creating abilities. However able they might appear, they have a lot more to learn. It is important that your planning supports the development of pupils' computational thinking and programming skills as they carry out a computing project within your subject area.

Resourcing

Programming languages

Each of the major programming languages used at KS3 has strong arguments in its favour. However, we must remember that teaching programming should not only help pupils learn a particular language but also teach them the underlying skills, such as sequencing, selection and repetition.

Which programming language should I use?

Consider the following.

- **How well do you know the language?** You need to support pupils from the least to the most able. Choosing a language you know well, or can learn quickly, makes you better prepared to answer questions and solve problems, as well as to suggest ways to help and extend the learning of your most able pupils.
- **What prior learning have pupils had?** If pupils have had opportunities to develop programming skills in a particular language, then it might be appropriate to consider using this language. However, it is the underlying skills that are important, so if you are more familiar with a different language, you may consider giving pupils the opportunity to transfer the skills they already have developed to a different language.
- **Are there high-quality resources and a community to support teaching and learning?** Finding published, differentiated resources makes it easier to plan great lessons and support your pupils.
- **Can pupils easily access the language at school?** There is a wide range of programming languages available within the Programming Environment on the C2K managed service. If your school uses this service (or another online programming environment), using one of these languages may be the best option, as installing another language on a managed network can be problematic.
- **Can pupils easily access the language at home?** You will almost certainly have pupils who want to do more programming at home. Is there a free, portable and/or web-based version of the language you are using so they can develop their interest outside school?

Which computing languages are available?

There is a broad range of languages available. However, at KS3 it would be expected that pupils will gain experience of using at least one language that is textual.

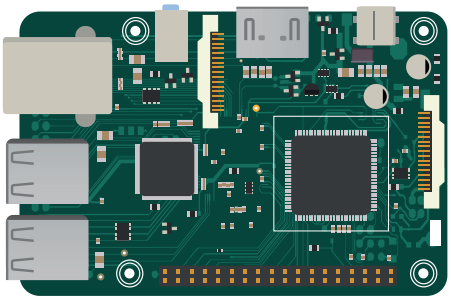
- Many pupils have experience of visual languages such as Scratch and Kodu from primary school. This is an excellent starting point, as side by side comparisons allow pupils to swiftly transfer to a textual language. At KS3 they are expected to gain experience of using at least one language that is textual.

- General-purpose textual languages such as Python, Visual Basic.NET, C++, C#, Ruby and Java are available in the C2K Programming Environment. They allow for a wide range of project work, including graphics, apps and games.
- Greenfoot is an interactive Java development environment designed primarily for educational purposes. It allows easy development of two-dimensional graphical applications, such as simulations and interactive games.
- Several languages (such as Logo) control the behaviour of a turtle or robot while at the same time introducing the idea of a textual language. Many general-purpose languages, including Python, also offer this functionality.
- Javascript is a fully featured programming language and can be used to teach programming.
- Mediator is a multimedia authoring package available on the C2K managed network, which allows pupils to program using a textual scripting language.

HTML and CSS are examples of specialised and declarative text-based mark-up languages. They are not appropriate choices for solving computational problems.

Hardware

Most school computer rooms can provide the hardware and software necessary for pupils to develop computing skills.



Small computers such as the **Raspberry Pi** were specifically developed for educational use and are excellent teaching tools. Pupils can see individual hardware components and the Raspberry Pi offers access to operating system functionality, such as the command line that might be restricted in school computer rooms. The command line allows pupils to give the computer textual commands and to query details such as network settings. The Raspberry Pi is valuable in areas such as Science and Technology where such 'low-level' access can allow external devices such as sensors or electronic circuits to be easily interfaced. Consider carefully how to incorporate small computers such as the Raspberry Pi into your teaching and the extra costs involved in connecting a monitor, keyboard, mouse and power supply. More information can be found at https://www.raspberrypi.org/

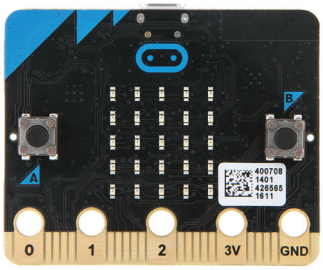
Arduino microcontroller development boards are also a good option for subjects such as Technology and Design and can be readily programmed using a dedicated programming environment. A range of 'shields' allow the capability of the boards to be expanded and more enthusiastic pupils can develop their own shields. More information can be found at www.arduino.cc/

BBC micro:bit

The micro:bit is a small computer board that has been released to all Year 8 pupils in Northern Ireland (and the equivalent year group in the rest of the UK). It has a wide range of features, including:

- 25 LED display (can be used to display patterns or scrolling text);
- 2 programmable buttons;
- compass;
- accelerometer;
- input/output pins to connect to external devices; and
- Bluetooth connectivity.

It has been designed to encourage children to get actively involved in programming and in creating everything from robots to musical instruments. A range of programming languages is available online at www.microbit.co.uk/ and loading programs onto the device is a simple process. In addition, the micro:bit can be programmed via Bluetooth from a mobile phone. Apps can be downloaded for both Android and iOS phones.



urbanbuzz/Shutterstock.com

There are many opportunities for the micro:bit to be used in a wide range of subjects and some examples are given later in this guide. Schools can purchase extra micro:bits after the initial free launch, so resources produced are not just be applicable to a single year group.

Software

There are many IT tools to support the development of computing skills, such as software development environments, animation tools, simulators and emulators.

Consider their suitability for your project.

- Will they do what you need them to?
- Are they easy to use?
- Will all pupils be able to access them away from school?

All schools with access to the C2K managed network can give their pupils access to the Programming Environment. A wide range of software development environments is available within this.

Unplugged

Don't forget that many of the concepts of computational thinking can be taught without any hardware or software at all. If computer access is limited, spend some valuable time in planning. Pupils can develop their ideas using pseudocode or flowcharts. In fact, good planning shortens the time needed to complete a project and can result in a higher quality solution better suited to the intended audience and purpose.

Teaching

In his theory of learning, Papert states that people learn best through making things for others.

Pupils learn more when they write about a topic than when they read about it, especially if they know that you, and perhaps others, will be reading what they write. It seems likely that this is true of every aspect of computing.

- Pupils learn computing skills more effectively by writing programs.
- Pupils learn more effectively if they're doing something creative, such as writing a program to solve a problem for someone.
- Pupils develop a richer digital literacy if they document what they know and learn from others through blog posts, audio recordings or screencasts.

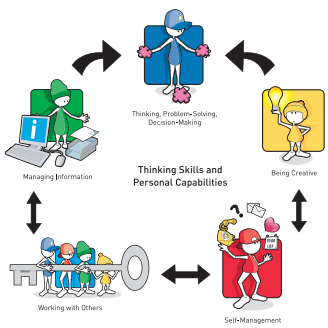
When teaching and developing computing skills, look for practical, creative projects for pupils to work on individually or in groups, ideally bringing together Computer Science and Digital Literacy. Projects are more motivating if there is a clear end product. Choose topics that naturally arise from the learning and teaching in your subject. In that way, they also develop pupils' understanding of subject knowledge. Also, look for a real audience and purpose for pupils' work, whether they're creating software or digital content for younger pupils or developing a solution to a real-world problem. This ties in closely with Express at higher levels, where pupils 'create, present and communicate their information and multimedia products for specific audiences and purposes'.

Project work is sometimes taught in a very formal way, with a rigid development structure. This 'waterfall model' has its strengths, not least of which is the structure it provides to the inexperienced and the less able. However, it also has significant weaknesses and does not guarantee success. Experienced developers tend to adopt more flexible approaches, where multiple prototypes are designed, created, tested, evaluated and improved. This is commonly known as agile development. Introducing this way of working in the classroom makes computing projects more enjoyable and helps to prepare pupils for their future professional lives. This aligns closely with the Thinking Skills and Personal Capabilities of the Northern Ireland Curriculum, where teachers support pupils in developing skills such as Being Creative, Working with Others and Self-Management.

As our Thinking Skills and Personal Capabilities microsite explains:

The curriculum emphasises the development of pupils' skills and capabilities for lifelong learning and participating in society. By engaging pupils in active learning contexts across all areas of the curriculum, teachers can develop pupils':

- personal and interpersonal skills;
- capabilities and dispositions; and
- ability to think both creatively and critically.



http://www.nicurriculum.org.uk/curriculum_microsite/TSPC/

Teaching programming

For some pupils, the fact that there are often several possible answers to a problem can be daunting. Others aren't used to the 'rapid fail – correct – fail better' model of computer programming.

Create a classroom environment of mutual respect and acceptance in which people learn through their mistakes. It is not unusual for professional programmers to spend over 50 percent of their time locating and fixing mistakes in their programs. This can be very challenging for the novice, and it is important to teach pupils techniques for locating and correcting their mistakes. This is not the same as testing, which can tell us that the program does not do what we intended, but cannot tell us why.

When pupils begin programming, they often need assistance in debugging. This can quickly become chaotic if their default is to immediately ask the teacher for help. One way to alleviate this problem is to implement a 'brain, buddy, book (or internet), then teacher' model, where pupils can only seek help from the teacher once they have exhausted the other routes of support. This is in line with the requirements of Evaluate, where pupils 'talk about, review and make improvements to work, reflecting on the process and outcome and consider the sources and resources used, including safety, reliability and acceptability'.

Encourage pupils to show their understanding by explaining their code line by line to one another. This is called rubber ducking or rubber duck debugging. An internet search should bring up some examples and give more information about this technique.

You will not (and need not) know the answers to all the questions raised by pupils in a computing classroom. It is important that pupils see you using strategies to debug program code, to find answers and to model different possible solutions.

Let your pupils explore. Much learning happens through guided exploration. Giving pupils the basic instructions to change the colour of text or create simple graphics allows them to customise tasks and put their stamp on their work, even when you are only asking for simple functionality such as working out the average of some numbers.

Inclusion

The digital age has seen the web, interactive whiteboards, virtual learning environments, video conferencing, blogs, wikis, podcasts, video and mobile devices have a transformative impact on both learning and teaching. Using technology draws on and enhances pupils' digital skills, and has opened up subject areas previously unavailable to many pupils. The following section is based on Naace/CAS (Computing at School) joint guidance. See the further reading section at the end of this guide for more details.

The digital divide

Pupils in your school probably come from a range of backgrounds, with access to digital technology influenced by social, cultural and economic factors. Computing can be used as a vehicle for social mobility, with those who excel being in high demand across large parts of the economy.

Huge numbers of job opportunities exist in this sector. When selecting resources and technologies to deliver computing, take care to ensure that all pupils have the opportunity to study outside the classroom and become independent learners. Ways to provide access include running after-school clubs, having computers and software in libraries, and using licensing agreements or open source software so that resources can be used at home.

Gender

Computing can appeal to all pupils. Take care to counter stereotypes within school (and society in general) that computing is a male-only field. There are many organisations supporting women in technology, for example see http://casinclude.org.uk/. Highlight the positive contributions of female role models such as Ada Lovelace, Grace Hopper, Jeannette Wing and Dame Wendy Hall. Lesson examples and project topics should be carefully considered to appeal to both genders. Be wary of pursuing activities that appeal to one gender or another – for instance, certain types of computer game.

Assistive technology

Computing can be made accessible to pupils with special educational needs or disabilities (SEND) through the use of assistive technologies, including hardware and software. Examples include adapted mice or keyboards, Braille displays, screen readers and adjusted system settings for dyslexia. These technologies can also be applied to other subjects, allowing access across the curriculum. When pupils are designing digital artefacts they should consider building in support for SEND users, for example through the use of colour schemes, layouts and support for screen readers.

English as an additional language (EAL)

Computing offers a range of tools for EAL pupils. Many software products and websites used in the classroom have in-built internationalisation settings to allow pupils to use them in their first language. Machine translation of documents and websites may also be used, although the accuracy of translation cannot be relied upon.

Gifted and talented pupils

Computing is an activity where pupils often voluntarily spend a significant amount of time outside school learning independently. It might be difficult for you to stay ahead of your pupils in all aspects of computing, and pupils may show knowledge and skills beyond that expected.

As a teacher, it is important that you encourage pupils who are displaying exceptional and esoteric skills to share their knowledge with others. You can seek out advice from subject support groups such as CAS and Naace on how to guide them. For example, you might advise a programmer with an interest in Maths to look at extended tasks on Project Euler (http://projecteuler.net), or to teach themselves a functional programming language, such as Haskell. Students can find out more at https://www.haskell.org/ Your role is to structure and facilitate learning, guiding the pupil to relevant material and external support.

Lesson ideas

The following lesson ideas have been selected because they all use software and hardware readily available in the classroom. They are meant to be a starting point from which full projects can be developed. There is no single 'correct' solution to any of the activities and pupils should be allowed to explore a range of solutions. It is through this experimentation that they will develop their programming skills.

Title	Motion Alarm
Subject	Technology and Design
Resources	BBC micro:bit with external speaker attached MS Block Editor MS Touch Develop
Description	The basic program will make use of the accelerometer to sound an alarm if the micro:bit is moved. The program can be developed to allow the alarm to be disabled/reset using a sequence of button presses. This is a good activity to allow transition from visual programming in Block Editor to textual programming in Touch Develop. The more advanced alarm functions will require the more advanced tools of Touch Develop.

Title	Drawing Polygons
Subject	Mathematics
Resources	PCs running Python with graphics library
Description	Given the number of sides as input, the program will draw the corresponding polygon and also output the calculated internal angle.

Title	Numbers/Letters Game
Subject	Mathematics or English
Resources	PCs running Gamemaker
Description	Design a simple game for children. Player moves character around the screen collecting numbers or letters, for example vowels or multiples of 7. Points are scored for collecting correct objects and lost for collecting incorrect objects. The game can be extended to cover, for example a range of multiplication tables. Before playing, the user selects the appropriate number.

Title	Compass App
Subject	Geography
Resources	BBC micro:bit
Description	Program device to get magnetic heading from internal compass. When the user presses button A, the heading in degrees will be displayed on the LEDs. When the user presses button B, the heading will be displayed as cardinal/ordinal point, in other words N, NE etc. Some extra complexity is involved in this task as pupils have to determine how best to calculate the correct cardinal/ordinal point. This will require one of the more capable textual languages such as MS Touch Develop, JavaScript or Python. All are available online. See www.microbit.co.uk

Title	Graphing Functions
Subject	Mathematics
Resources	PC running Python with TKInter and Plotly libraries
Description	The user selects a function from a list displayed on screen. The program then plots that function as a line graph. TKInter is Python's de-facto standard GUI (Graphical User Interface) package and allows programmers to design simple user interfaces. Although plotting graphs with labels and so on is a complex task, the use of the Plotly library greatly simplifies the process making it feasible at this level. Full details on Plotly can be found at https://plot.ly/

Title	Modelling Populations
Subject	Science
Resources	PC running Greenfoot
Description	Simulations, such as modelling populations are a particular strength of Greenfoot. Various parameters can be programmed and the simulation allowed to run to observe what happens to populations. You can find good examples by searching for scenarios at www.greenfoot.org Go to the home page and select the simulation tab.

Title	Temperature Control
Subject	Science
Resources	BBC micro:bit with motor driver circuit (depending on motor)
Description	The temperature of the room can be measured using the on-board temperature. If the temperature is too low, then a heater is turned on (simulated by lighting up LED display). If the temperature is too high, then a fan is switched on, or alternatively this can again be simulated by an X on the LED display. This program provides an opportunity to introduce a number of more complex concepts, including feedback loops and hysteresis. If a real fan is to be used, then a motor driver circuit may be required. Such extension boards are available commercially. This will allow further advanced programming techniques to be explored, as it is possible to vary the fan speed using a PWM output on the micro:bit.

Title	Interactive Storybook
Subject	English
Resources	PC running Mediator 9.0
Description	Pupils will be aware of the wide range of children's interactive storybooks available on Apple and Android tablets. Using Mediator, it is possible to create a similar product. Voiceovers and videos can be incorporated and interactive features can be added to pages. Mediator uses event-driven programming, where events such as mouse clicks, timers and drop-ons determine the flow of the program. The programming interface can be semi-graphical or it can be switched to textual for more flexible programming.

Title	Programming PIC devices
Subject	Technology and Design
Resources	PC with attached PIC board Software such as PICAXE Editor
Description	There is a wide range of activities possible, due to the flexibility of the PIC devices. Projects such as steady-hand games, and electronic dice can be programmed. Programming is done through creating flowcharts in Logicator, part of the PICAXE Editor package. Flowcharts are a particularly useful tool when exploring Boolean logic and how it affects program flow.

Title	Vocabulary Quiz
Subject	Modern Languages
Resources	PC running Mediator 9.0
Description	The multimedia features of Mediator allow it to work well in modern languages activities. Correct pronunciations of words can be recorded. Quizzes can be created in a number of formats and scores can be saved in a file to allow users to check progression.

Title	Calculating Prime Numbers
Subject	Mathematics
Resources	PC running a general-purpose language such as Python, Java or C
Description	This is a classic programming problem in mathematics. While the concept is simple, there are a number of possible algorithms that can be used to solve the problem. Pupils can explore several to compare them. One of the most efficient is the gloriously named the sieve of Eratosthenes, and exploring how this algorithm works is an interesting mathematical topic in itself. Solving this problem also requires the use of more advanced data structures such as arrays. Programs can start with asking the end user how many prime numbers are to be generated. Additional features can then be added, such as checking if an input number is a prime number.

Title	Key Words Dictionary
Subject	Any Subject
Resources	PC, App Inventor and an Android phone or tablet
Description	App Inventor is a Scratch-like programming environment for designing apps for the Android environment. It is very accessible for teenagers, with the added incentive that they get an app they can load onto their phone. Being a graphical programming environment, complex apps can quickly overtake screen space. However, a dictionary app, even one including pictures is a very accessible activity within any subject area. The end product also provides a useful revision resource.

Resources and further reading

Here is a small selection of resources for computing at KS3 and KS4. You can find more detailed lists by selecting the resources tab on the computing at school website http://www.computingatschool.org.uk/

Background

Computing at School Working Group, *Computer Science: A Curriculum for Schools* (Computing at School, 2012). See http://www.computingatschool.org.uk/

Department of Education – Teach computing. Go to the Get Into Teaching website https://getintoteaching.education.gov.uk/ and type 'computing' into the search facility.

Helsper, E J and Eynon, R, 'Digital Natives: Where is the Evidence?' *British Educational Research*, (2010) 36 (3), 503–520.

Papert, S, *Mindstorms: Children, Computers, and Powerful Ideas* (Basic Books, 1993).

The Royal Society, *Shut Down or Restart? The Way Forward for Computing in UK Schools* (London, 2012). Go to the Royal Society website https://royalsociety.org/, select the **Topics and policy tab**, then **Reports** and type in 'computing', or 'shut down or restart' into the search facility.

Rushkoff, D, *Program or be Programmed: Ten Commands for a Digital Age* (OR Books, 2009).

Subject knowledge

Bentley, PJ, Digitized: *The Science of Computers and How it Shapes our World* (Oxford University Press, 2012).

Berners-Lee, T, 'Answers for Young People'. Go to the W3 website http://www.w3.org/ Select the tab **About W3**, select **Tim Berners-Lee**, then select **Kids' Questions**.

Brennan, K and Resnick, M, 'New Frameworks for Studying and Assessing the Development of Computational Thinking' (2012), available at: http://web.media.mit.edu/~mres/papers.html

Computing at School, *The Raspberry Pi Education Manual* (Computing at School, 2012), available from the STEM learning website: https://www.stem.org.uk/ Select **RESOURCES**, then enter 'raspberry pi' into the search facility.

Kemp, P et. al. (2011–) *A-level Computing* (Wikibooks), available at: https://en.wikibooks.org/wiki/A-level_Computing/AQA

O'Byrne, S and Rouse, G, *OCR Computing for GCSE* (Hodder Education, 2012).

Extended learning and competitions

The British Informatics Olympiad: a computer programming competition for pupils under 19. Finals of the competition take place in Cambridge: see www.olympiad.org.uk

CoderDojo: organisation promoting computer programming and technology; locations spread across the UK: see https://coderdojo.com/

Make Things Do Stuff: campaign and website providing pupils with links to clubs, communities, competitions and events; provides online projects: see www.makethingsdostuff.co.uk

RaspberryJam: monthly meeting for Raspberry Pi enthusiasts of all ages; locations spread across the UK: see www.raspberrypi.org/jam/

UK Schools Computer Animation Competition: run by the University of Manchester, open to UK pupils aged 7–19. Go to the university's website http://www.manchester.ac.uk/ and type 'computer animation competition' into the search facility.

Young Rewired State: an organisation promoting computer programming through online networking and free camps of varying lengths for pupils aged 18 and under; locations spread across the UK: see www.youngrewiredstate.org

Teaching resources

New Zealand-based **Computer Science (CS) Unplugged** produces an excellent collection of resources exploring computer science ideas through classroom-based, rather than computer-based, activities: see http://csunplugged.org/

Computing at School (CAS) hosts a large resource bank of plans, resources and activities. CAS is free to join: see www.computingatschool.org.uk

Naace (the ICT association) and **CAS** have developed joint guidance on the new computing curriculum. Visit the Naace website https://www.naace.co.uk/ and select the **Publications** tab, then filter by **Computer Education**.

A group of teachers and teacher trainers convened by the National College for Teaching and Leadership (NCTL) worked together to curate **resources for initial teacher training** for the computing curriculum, many of which may be useful for continuing professional development and classroom use: see https://sites.google.com/site/primaryictitt/

The 2008 Royal Institution Christmas Lectures were given by computer scientist Chris Bishop. Visit the website http://www.richannel.org/ then select the **Christmas Lectures** tab.

Excellent resources are available for teaching with MIT's Scratch programming toolkit, together with an online support community, on the ScratchEd site: see http://scratched.media.mit.edu/

Resources for teaching safe, respectful and responsible use of technology are widely available. Good starting points for exploring these topics are www.childnet.com/teachers-and-professionals and https://www.thinkuknow.co.uk/teachers/

Glossary

abstraction (process): the act of selecting and capturing relevant information about a thing, a system or a problem.

abstraction (product): a representation of a thing, a system or a problem that contains only selected (relevant) details about it; for example, a diagram is an abstraction.

algorithm: a set of unambiguous rules or instructions to achieve a particular objective.

array: a data structure comprising a collection of values of the same type, accessible through an index.

assembly code: a human-readable programming language in which each instruction corresponds to a single executable instruction for a CPU.

binary: a method of encoding data using two symbols, 1 and 0.

binary number: a number written in the base 2 number system.

bit: a basic unit of data that stores one binary value, 1 or 0.

bitmap: a collection of pixels forming an image.

Boolean: a data type with only two values, TRUE or FALSE.

browser cookie: a small piece of text recording activity about websites you visit, stored on your computer.

circuit: a grouping of electronic components that allow for operations to be performed.

code: any set of instructions expressed in a programming language.

coding: the act of writing computer programs in a programming language.

colour depth: the number of different colours that may be used in an image, dictated by the number of bits used to represent the colour of each pixel.

compiler: a program that converts programs written in one language (source code) into equivalent programs written in a different language (often in the form of instructions that a processor can execute).

computational thinking: a philosophy that underpins computing through decomposition, pattern recognition, abstraction, pattern generalisation and algorithm design.

CPU: central processing unit; the device within a computer that executes instructions.

data structure: a particular way to store and organise data within a computer program.

debugging: the process of finding and correcting errors in programs.

decimal: the base 10 number system.

decomposition: breaking a problem or system down into its components.

digital: using discrete binary values.

digital artefact: digital content made by a human with intent and skill.

digital creator: a person who makes digital artefacts.

digital media: media encoded in a computer readable form.

hardware: the physical components that make up a computer.

HTML: hypertext mark-up language; the language used to create web pages.

input (noun): an input is a data value passed from the outside world to a computer.

input (verb): to input is to send data from the outside world into a computer system.

internet: a network of interconnected networks.

interpreter: a program that converts instructions written in one language into equivalent instructions in another language, and executes each instruction as soon as it is translated.

IP address: Internet Protocol address; a unique numeric value that is assigned to a computer or other device connected to the internet so that it may be identified and located.

LAN: Local Area Network; a network of computers covering a small area, such as a building or small group of buildings.

lists: a data structure for storing ordered values.

model: a representation of (some part of) a problem or a system.

modelling: the act of creating a model.

modular design: the practice of designing a system or program as a set of independent but interacting units (modules) that may be implemented and tested separately before bringing them together to solve the overall problem.

network: more precisely, a computer network; a collection of computational devices (personal computers, phones, servers, switches, routers, and so on) connected to one another by cables or by wireless media and arranged so that data may be sent between devices either directly or via other devices.

operating system: a set of programs that manage the functioning of, and other programs' access to, hardware.

output (noun): a response from a system.

output (verb): to generate an output.

packet: more precisely, a network packet. A formatted unit of data for transmission across a network. Each packet contains part of a message plus some additional data, including where it is from and where it is going.

pixel: the smallest controllable element of picture/display.

process (noun): a process is a running program.

process (verb): the act of using data to perform a calculation or other operation.

program: a set of instructions that the computer executes in order to achieve a particular objective.

programming: the craft of analysing problems and designing, writing, testing and maintaining programs to solve them.

programming language: formal language used to give a computer instructions.

repetition: the process of repeating a task a set number of times or until a condition is met.

resolution: a measurement of the number of pixels needed to display an image.

router: more precisely, a network router. A router is a device that connects networks to one another (typically one or more local area networks (LANs) to a wide area network (WAN)), and directs packets between networks. A home broadband router performs the functions of a switch while allowing computers to connect to the internet.

selection: using conditions to control the flow of a program.

sequence (noun): an ordered set of instructions.

sequence (verb): to arrange a set of instructions in a particular order.

server: a computer or program dedicated to a particular set of tasks that provides services to other computers or programs on a network.

software: the programs that run on the hardware/computer system.

switch: more precisely, a network switch. This is a device that connects multiple computers to one another on a single local area network (LAN), and directs packets from machine to machine.

table: a data type storing organised sets of data under column headings.

Unicode: a standardised system for representing individual characters as sequences of bits.

variable: a data store used in a program.

web browser: a computer program to view websites.

WAN: Wide Area Network; a network of computers covering a wide area, consisting of two or more interconnected local area networks. The internet is the largest example of a WAN.

World Wide Web: a service made of connected hypertext documents linked together across the internet.

COUNCIL FOR THE **CURRICULUM, EXAMINATIONS** AND **ASSESSMENT**
29 Clarendon Road, Clarendon Dock, Belfast BT1 3BG
Tel: +44(0)28 9026 1200 Fax: +44(0)28 9026 1234
Email: info@ccea.org.uk Web: www.ccea.org.uk

Rewarding Learning